File Number 1620-25
Form C26-5619-1

IBM Systems Reference Library

1620 FORTRAN (with FORMAT)

FORTRAN is an automatic coding system that allows the engineer and scientist to utilize a computer for problem solving, with only a little knowledge of the computer and a short period of training. The manual is divided into six sections, each developed for a specific need of a FORTRAN user.

Part 1, Introduction to IBM FORTRAN
Part 2, Writing the 1620 FORTRAN Program
Part 3, Operating Principles
Part 4, Analysis of the FORTRAN Program
Part 5, The FORTRAN Pre-Compiler Program
Part 6, Summary of 1620 Operating Principles

This manual is written for the following IBM Applied Programs:

1620-FO-003	FORTRAN with FORMAT for paper tape
1620-FO-004	FORTRAN with FORMAT for cards
1620-FO-005	FORTRAN Pre-Compiler for paper tape
1620-FO-006	FORTRAN Pre-Compiler for cards

This manual contains minor changes to the previous edition, C26-5619-0, and incorporates changes released in Technical Newsletters N26-0010 and N26-0020. While the format has been changed to conform to that of the Systems Reference Library, the original publication and applicable newsletters are not obsoleted.

Copies of this and other IBM publications can be obtained through IBM Branch Offices. Address comments concerning the contents of this publication to IBM, Product Publications Department, San Jose, California

Contents

Preface

Each of the six sections of this manual is developed for a specific need of a FORTRAN user. This preface was designed to enable you to quickly locate and extract the segments of 1620 FORTRAN that are most important to you.

Part 1. INTRODUCTION TO IBM FORTRAN is intended for readers who have neither a previous knowledge of other FORTRAN systems nor a background in data processing. This part tells what FORTRAN is, and what the 1620 Data Processing System is.

Part 2. WRITING THE 1620 FORTRAN PROGRAM is developed primarily for the "nonprofessional programmer," a person not engaged in programming as a full time occupation. This part of the manual tells how to write a FORTRAN program. If your responsibilities are concerned with only **writing** FORTRAN programs, and not **processing** them on the computer, you need not read the other parts of the manual. Appendix A contains a summary of 1620 FORTRAN statements.

Part 3. OPERATING PRINCIPLES provides the information necessary to implement the FORTRAN system on the 1620 computer. If you are a machine operator, or a programmer processing a program, this part of the manual will show you how to place the program into the machine, provide the proper setting of the switches, explain the use of the keys on the 1620, explain the type of programming errors that the FORTRAN program will detect, and show you how to enter input data.

Part 4. ANALYSIS OF THE FORTRAN PROGRAM is intended for the experienced programmer. This part of the manual describes certain features of the program, shows where data is located during processing, shows how the program may be amended, and provides the general format for card and paper tape input data.

Part 5. The FORTRAN PRE-COMPILER program is described in this part. The Pre-Compiler is a special program provided by IBM to enable the FORTRAN programmer to "pre-test" FORTRAN programs. This program detects and permits corrections of the more common programming errors. Read Parts 1 and 2 before reading this part of the manual.

Part 6. Appendix B is a **summary** of the operating principles described in the IBM *Reference Manual, 1620 Data Processing System* (Form A26-4500). This part of the manual is intended for the reader that processes FORTRAN programs and has no prior 1620 operating experience. This part is included in the manual in order to provide the FORTRAN user with one manual that contains all information necessary for the utilization of 1620 FORTRAN.

Part 1—Introduction to IBM FORTRAN

FORTRAN (FORmula TRANslation) is an automatic coding system that allows the engineer and scientist to utilize a computer for problem solving with only a slight knowledge of the computer and a short period of training.

FORTRAN is written in a language that is a compromise between the language of the computer and the language of the engineer and scientist. To satisfy the computer, symbols are used that the computer can understand and this requires that the rules for their use be closely followed. To satisfy the engineer and scientist, as many of the detailed computer control operations as possible are eliminated from the job of writing programs, and a problem statement format close to that of the mathematical equation is used.

FORTRAN programs are written on paper coding forms, punched into IBM cards or IBM paper tape, and then processed on an IBM Data Processing System. This manual is written for the IBM 1620 Data Processing System, a low-cost, solid state digital computer.

Digital Computers

A digital computer is composed of the following elements:

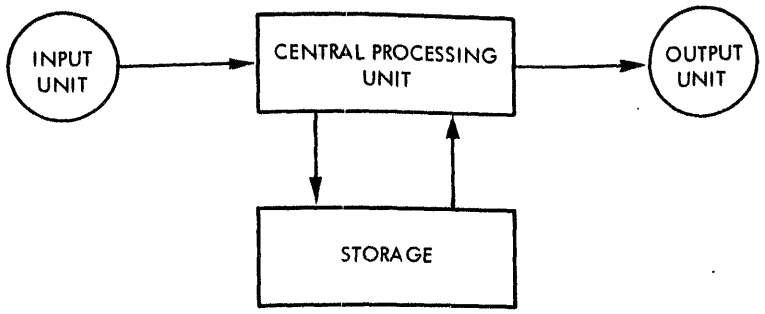
1. *Input Unit.* Digital computers accept numbers, letters, and symbols. Information can be fed into the system by using punched cards, punched paper tape, or by inserting information manually through a typewriter keyboard.

2. *Central Processing Unit.* The sequence of steps to be performed must be translated into detailed instructions which the computer can understand. A series of instructions is called a **program.** When it is retained in a storage device, it is called a **stored program.** These coded instructions in storage are available as needed to direct and complete an entire sequence of operations. Special instructions may permit logical-arithmetic decisions to be made based on intermediate results; these decisions allow the computer to select the proper course among several alternatives for solving a problem. A logical-arithmetic unit can add, subtract, multiply, divide, and compare numbers in a manner similar to a desk calculator, but at lightning speed. Complex calculations are usually combinations of these basic operations. The logical-arithmetic unit can make **logical** decisions. It can distinguish positive, negative, and zero values and transfer this information to other units of the computer.

3. *Storage Unit.* Data can be internally stored until needed. This information is stored in a manner quite similar to the way music or speech is stored on a tape for playback on a tape recorder, although the notation used is quite different. Stored information can be referred to once or many times, and can be replaced whenever desired. The information stored by the computer can be original data, intermediate results, reference tables, or instructions. Each storage location is identified by an individual location number which is called an **address.** By means of these numerical addresses, a computer can locate data and instructions as needed during the course of a problem.

4. *Output Unit.* While doing its work, the computer can produce answers in several forms. Results may be punched into cards, paper tape, or printed in report form.

The organization of these elements to form a computer may be illustrated as follows:



The elements of a computer function in a manner which may be compared to the steps required for solving a problem by paper and pencil methods. Input corresponds to the information given in the problem. The rules of arithmetic control the handling of the problem. The logical-arithmetic functions are the same as the functions of manual calculations. Storage may be compared to the work papers on which intermediate answers are noted. The answers are the output.

The Stored Program

"Program" is just another way of saying "series of instructions and fixed data." A program must define in complete detail, for every conceivable combination of circumstances, just what the computer is to do with the data which will subsequently be fed into it.

An instruction may tell the computer what operation to perform and where to locate the data on which the operation is to be performed; another will tell what to do with the result. These computer instructions are stored in the sequence necessary to accomplish a given task, and form the stored program.

The various operations covered in these instructions are usually stated in a numerical or alphabetic code. Thus, the operations in a simple problem might be designated as follows:

Operation Code	Operation
21	add
22	subtract
26	store the result

These operation codes might be used in a stored program in the following manner:

	Operation Code	Storage Location	
Instruction #36	21	00879	00679
Instruction #37	21	00879	00659
Instruction #38	22	00879	00639
Instruction #39	26	01479	00879

Instruction #36 – tells the computer to add the number stored at location 00679 to the number stored in 00879.

Instruction #37 – add the number stored at location 00659 to the result in 00879.

Instruction #38 — subtract the number stored at location 00639 from the result in 00879.

Instruction #39 — store the result of the two additions and the one subtraction at location 01479.

The same program, coded in FORTRAN, might be:

$$D = A + X - Y$$

The complete solution of a problem, depending upon the type of problem to be solved, may require hundreds or even thousands of instructions. The computer can refer to them one after another, or it can be instructed to repeat, modify, or skip over certain instructions, depending on intermediate results or circumstances. However, such circumstances must be anticipated and appropriate instructions included in the program.

The ability to repeat operations combined with the ability to modify and skip over instructions permits a significant reduction in the number of instructions required to perform any given job.

The decision-making ability of the computer enables it to handle exceptions to standard procedures. Since a system will "remember" instructions for dealing with the exceptions, it can be made to handle automatically any situation that develops.

Up to this point, the computer has been treated as though it were a separate piece of equipment to be used by itself. However, in actual practice, the computer is used in conjunction with other equipment and with programming systems that are designed to aid the programmer in the preparation and operation of his programs. These total facilities for receiving information and producing desired results are called a data processing system. One part of such a system may be FORTRAN which is a programming system that enables a programmer to write a program with less effort than would otherwise be required. For the purpose of explanation, FORTRAN will be described in two parts: the FORTRAN System and the FORTRAN Language.

The FORTRAN System

The FORTRAN System consists of the following parts.

The Processor

The processor is a program developed by IBM. Its purpose is to tell the computer how to translate the FORTRAN language, written by the programmer, into the machine language used by the computer.

The Source Program

The source program defines the ultimate operations the computer is to perform and is written by the programmer in the FORTRAN language.

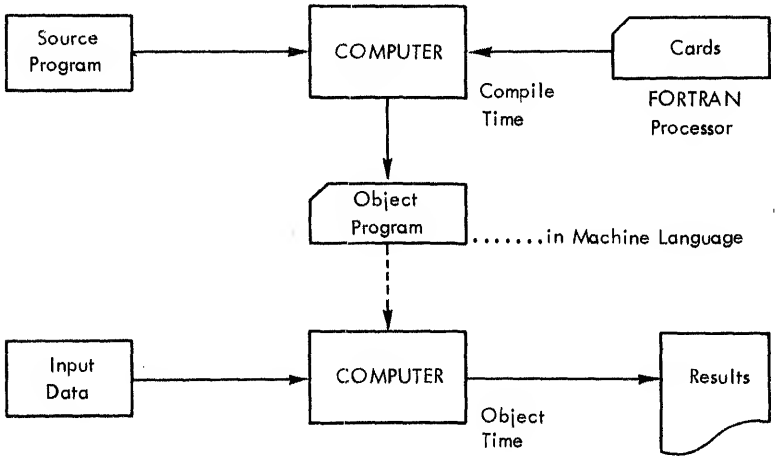
The Object Program

The source program, then, is input to the computer along with the FORTRAN processor. The computer follows instructions from the **processor** to convert the **source program** into a machine language which can then be run on the computer. This machine language program is called the **object program.** When the object program and the data to be processed is run on the computer to cause the desired computations, it is said to be **executed.** That is, execution is the actual operation of the computer while it is under the direction of the object program.

It is important in learning FORTRAN to remember the difference between the processor and the source program. The operation of converting the source program to an object program is referred to throughout this manual as **compilation,**

and events that occur at this time are referred to as occurring at **compile time.** The term **object time** refers to events that occur while the object program is being executed.

The diagram which follows illustrates this sequence of events.



The FORTRAN Language

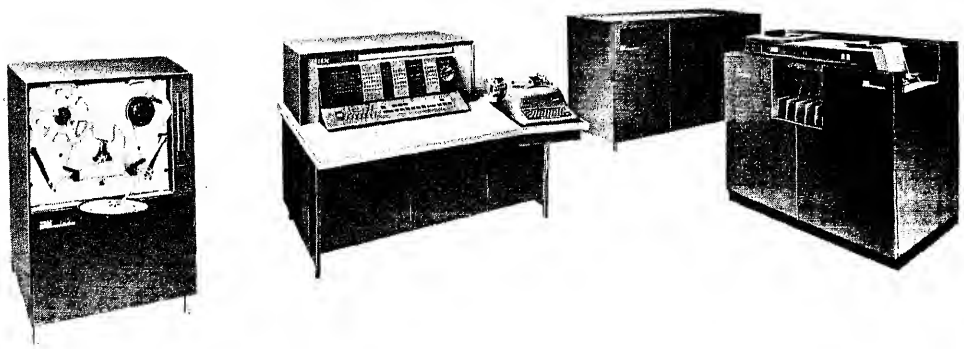
The FORTRAN language is composed of the individual commands or statements of a program consisting of operation symbols ($+$ or $-$), and expressions ($A + B - C$).

Statements are the sentences of the FORTRAN language. They may:

1. Define the arithmetic steps which are to be accomplished by the computer.
2. Provide information for control of the computer during the execution of the program.
3. Describe input and output operations which are necessary to bring in data and punch or write the results.
4. Specify certain additional facts such as the size of the input data that is read by the program.

The 1620 Data Processing System

The IBM 1620 Data Processing System is an electronic computer system designed for scientific and technological applications. The use of solid-state circuit components and the availability of from 20,000 to 60,000 positions of core storage provide the 1620 system with the capacity, reliability, and speed to solve problems that in the past have required the use of larger data processing systems.



Four units are available with the IBM 1620 Data Processing System. The IBM 1620 Central Processing Unit contains the computer, 20,000 positions of core storage, a console panel, and an input/output typewriter. Paper tape operations are permitted by the IBM 1621 Paper Tape Reader unit, which also includes the paper tape controls and the IBM 1624 Tape Punch. The IBM 1622 Card Read Punch is available for card operations. The IBM 1623 Storage Unit expands the 20,000 core storage positions in the Central Processing Unit to 40,000 or 60,000 positions.

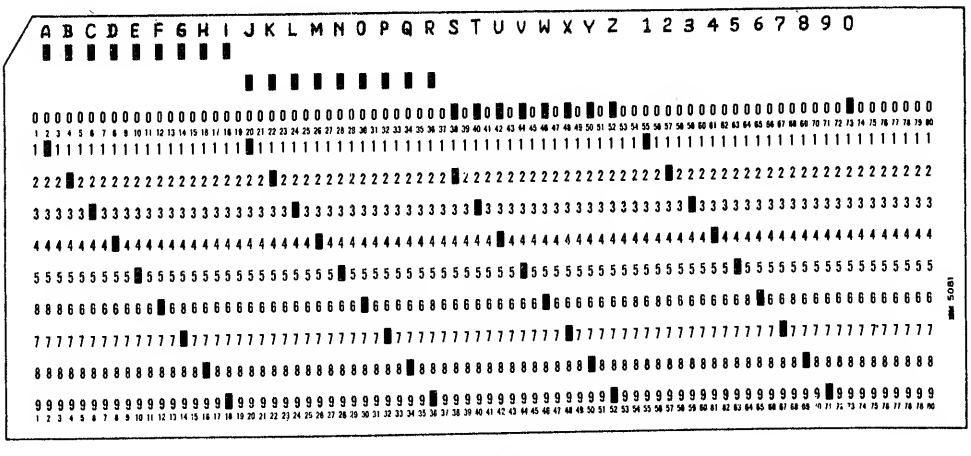
The console of the 1620 contains control keys, switches, an indicator panel, and a typewriter. The control keys and switches are used for manual or automatic operation of the system. The console panel provides visual indication of the status of various registers and indicators. The typewriter provides direct entry of data and instructions into core storage; it also provides a permanent log of the operator's intervention during the execution of a program.

Information is entered into the system by input devices; namely, the IBM 1621 Paper Tape Reader, the IBM 1622 Card Read Punch, and the typewriter. The 1622 reads 80-column cards at a maximum rate of 250 cards per minute. The 1621 reads an 8-track paper tape at the rate of 150 characters per second. The operator's typing speed determines the rate at which information enters through the typewriter.

The IBM 1622 Card Read Punch, the 1624 Tape Punch, and the typewriter are output devices which record the processed data. The typewriter prints at a maximum rate of 10 characters per second; the card punch and tape punch operate at the rate of 125 cards per minute, 15 characters per second, respectively.

The IBM Card

The IBM card is divided into 80 vertical areas called "columns" or "card columns." They are numbered from 1 on the left to 80 on the right side of the card. Each column is then divided horizontally into twelve punching positions. The punching positions are designated from the top to the bottom of the card by 12, 11 (or X), 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The punching positions for digits 0 to 9 correspond to the numbers printed on the card. Each column of the card is able to accommodate a digit, a letter, or a special character. Thus the card may contain up to 80 individual pieces of information. Digits are recorded by holes punched in the digit punching area of the card from 0 to 9.



As illustrated in the drawing, a combination of a zone punch and a digit punch is used to accommodate any of the 26 letters in one column.

A card is divided into segments called "fields." A field is a column or columns reserved for the punching of data of a specific nature. The field may consist of from one to 80 columns depending upon the length of the particular type of information.

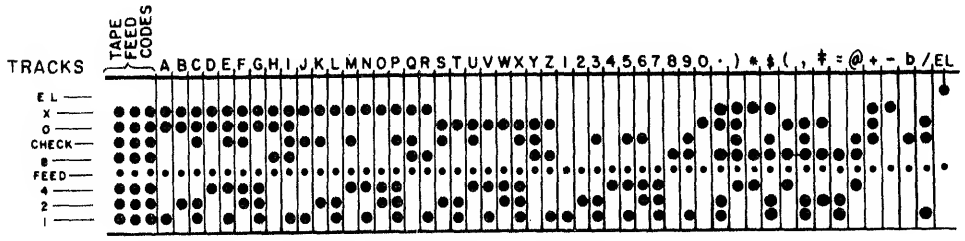
Paper Tape

Punched paper tape serves much the same purpose as punched cards. Developed for transmitting telegraph messages over wires between two computers, paper tape is now used for communication with other computers as well as for basic input to computers.

Eight-track paper tape has eight parallel tracks along which data can be recorded. One column of eight punching positions across the width of the tape is used to code numerical, alphabetic, and special characters.

The four lower tracks of the tape (excluding the feed holes) are called 1, 2, 4, and 8 and are used to record numerical characters. The numerical characters 0 through 9 are represented by a punch or punches in these four positions. The sum of the position values indicates the numerical value of the character: a hole in channel 1 represents a one; a combination of 1 and a 2 punch represents a three; and so on.

The X and O tracks are used in combination with the numerical channels to record alphabetic and special characters.



A parity check is made to ensure that each letter or number is punched correctly. This consists of automatically punching each column of the tape with an odd number of holes. The check channel is reserved for punching when the number, letter, or special character has an even number of holes. As the tape is read or punched, each column is checked to make certain that it is punched with an odd number of holes. For example, the basic 6-hole tape code for the letter A is coded for an odd number of holes, X-O-1, so the check hole is not punched. However, the letter C is coded X-O-1-2, which is an even number of holes, and the check code is punched to make an odd number.

Part 2—Writing the 1620 FORTRAN Program

This part of the manual explains how each FORTRAN statement is prepared and how they can be combined to solve problems in engineering and science. The terms Processor, Source Program, Object Program, and Compile Time are used throughout this part of the manual. If you are not certain of the meaning of each of these terms, you should review INTRODUCTION TO FORTRAN in Part 1.

In the previous section of the manual we learned that a FORTRAN program solves a problem by carrying out the instructions given by a series of statements. These statements can be arranged into four groups:

Input/output statements that read data into the program or print and punch the results of the program.

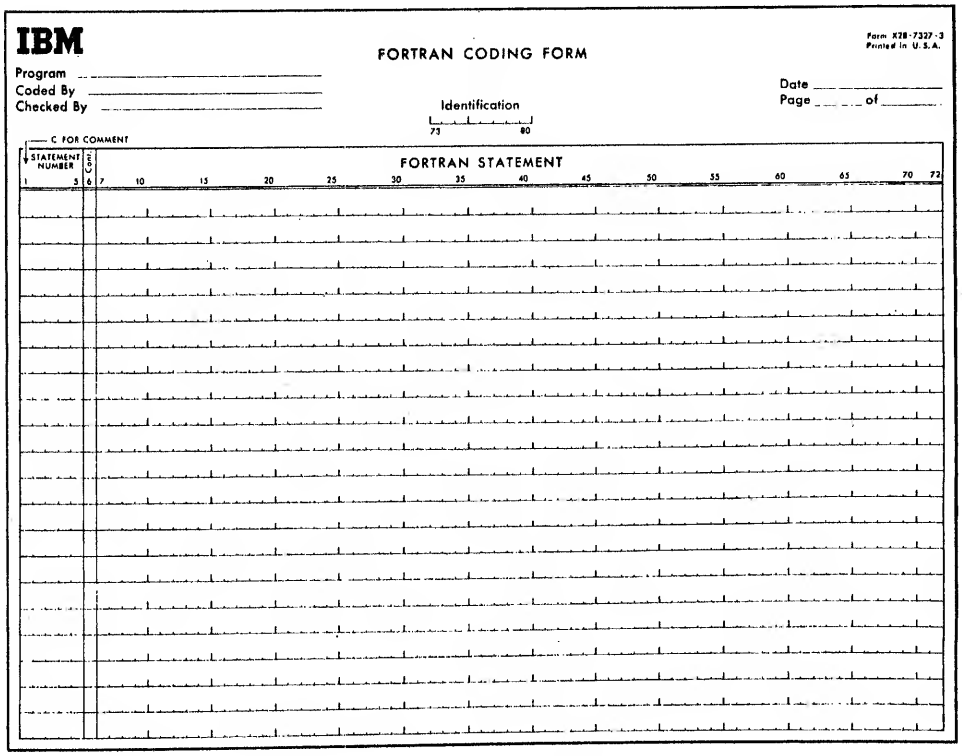
Control statements that may determine the sequence in which the statements will be followed or provide the program with the ability to deal with predefined exceptions to the procedure.

Specification statements that tell the FORTRAN program the amount and kind of input and output data it will process.

Arithmetic statements that specify the mathematical calculations to be performed.

The FORTRAN Coding Form

This coding form is used throughout the manual to illustrate examples of FORTRAN statements and it will be used when you write a FORTRAN program. For these two reasons, we will examine the coding form first.

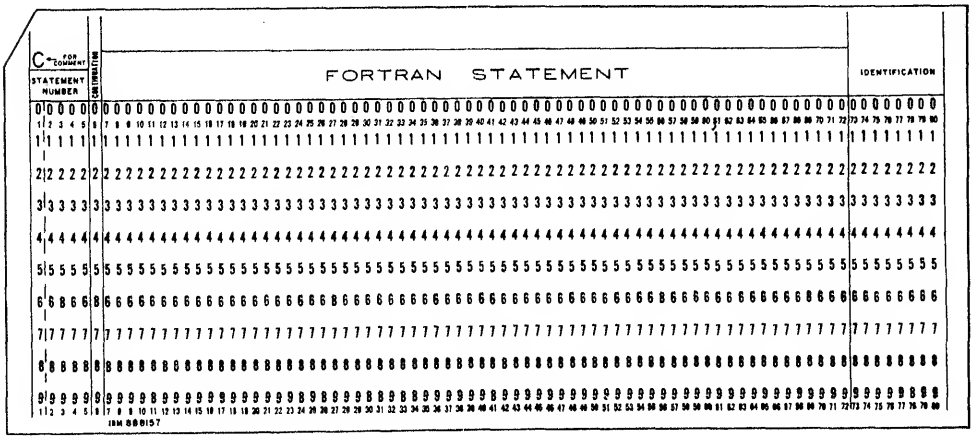


The coding form is used to record each statement of the program. The form contains 72 vertical columns. Columns 1 through 5 are used to write statement numbers. Statement numbers may be required by the program or may be used merely as a means of identifying statements. (The specific requirements of statement numbers are discussed later.)

Column 6 of the form is not used and must be blank. Columns 7 through 72 are used to write the statements. Each number, letter, and special character used in the program is written in a separate column of the form. Each statement must start and finish on one line. Statements cannot be continued from one line to another.

The FORTRAN Card

Each line of the coding form is punched into a separate card. The standard FORTRAN card is shown below.



Column 1 of the card is used to record the information contained in column 1 of the coding form. Column 2 in the card contains the information in column 2 of the form, etc. The information in all 72 columns of the coding form are punched into the card. Columns 73 through 80 are used for identification. The identification field is usually divided into two parts. The first part may contain a specific job number assigned to the problem by the programmer; the second part is used to assign a sequence number to each card in the program. The identification number "180024" in a card could mean it is the 24th card in the deck for job no. 18. This field, used at the option of the programmer, is of great value if cards from one program are mixed with other cards or are accidentally upset and their sequence lost.

Floating Point Arithmetic

Scientific and engineering computations frequently involve lengthy and complex calculations in which it is necessary to manipulate numbers that may vary widely in magnitude. To obtain a meaningful answer, problems of this type usually require that as many significant digits as possible be retained during calculation, and that the decimal point always be properly located. When applying such problems to a computer, several factors must be taken into consideration, the most important of which is the decimal point location.

Generally speaking, a computer does not recognize the decimal point present in any quantity used during the calculation. Thus a product of 414154 will result

regardless of whether the factors are 9.37 x 44.2, 93.7 x .442, or 937 x 4.42, etc. It would normally be the programmer's responsibility to be cognizant of the decimal point location during and after the calculation and to arrange the program accordingly.

The processing of numbers expressed in ordinary form (e.g., 427.93456, 0.0009762, 5382, —623.147, 3.1415927, etc.) can be accomplished on a computer only with extensive analysis to determine the size and range of intermediate and final results. This analysis and subsequent number scaling frequently requires a larger percentage of the total time needed to solve the problem than is required by the actual calculation. Furthermore, number scaling requires complete and accurate information regarding the bounds on the magnitude of all numbers that come into the computation (input, intermediate, output). Since it is not always possible to predict the size of all numbers in a given calculation, analysis and number scaling is sometimes impractical.

To alleviate this programming problem, a system is used in FORTRAN in which information regarding the magnitude of all numbers accompanies the quantities in the calculation. All numbers are represented in a standard, predetermined format which instructs the computer in an orderly and simple fashion as to the location of the decimal point. With this method, quantities which range from minute fractions having many decimal places to large numbers having many integer places may all be handled. This system is called "floating point arithmetic."

The notation used in floating point arithmetic is an adaptation of the scientific notation. That is, the decimal point of all numbers is placed to the left of the high-order (leftmost) nonzero digit. (This is often referred to as "normalizing" the number.) Hence, all quantities may be thought of as a decimal fraction times a power of ten.

$$427.93456 \text{ as } .42793456 \times 10^{3}$$
$$\text{and } 0.0009762 \text{ as } .97620000 \times 10^{-3}$$

where the fraction is called the mantissa, and the power of ten, indicating the number of places the decimal point was shifted, is called the exponent.

In floating point calculations, each quantity operated upon is expressed as a 10-digit number consisting of an 8-digit mantissa, and a 2-digit exponent. The magnitude of the number thus expressed must be zero or must lie between 10^{-100} and 100^{100}.

The mantissa consists of the leftmost eight digits of the floating point number. The decimal point is always assumed to lie immediately to the left of the high-order mantissa digit. The range of the mantissa is between .10000000 and .99999999.

The exponent represents the power of ten used to specify the location of the decimal point in the original number. The sign and magnitude of the exponent is determined by the number of places the decimal point is shifted in order to place it to the left of the high-order nonzero digit. The direction of shift determines the sign of the exponent; positive for left, negative for right.

The following examples demonstrate the conversion of numbers in ordinary form to a floating point notation.

Number	Floating Point Form
123.45678	$.12345678 \times 10^{3}$
.00765438	$.76543800 \times 10^{-2}$
—.12348693	$-.12348693 \times 10^{0}$
—.00000070	$-.70000000 \times 10^{-6}$

Fixed Point

Quantities used in a FORTRAN program may also be expressed in **fixed point** form. A fixed point number is an ordinary whole number, without a decimal point, consisting of the digits 0 through 9.

Constants and Variables

Mathematical problems usually contain some data that does not change throughout the entire problem, and other data that may change many times during calculation. These two kinds of data are referred to as "constants" and "variables," respectively. Both constants and variables can be used in FORTRAN if they are written so that the processor can distinguish one from the other.

Constants

A constant is any number which is used in computations without change from one execution of the program to the next. It appears in its actual numerical form in the source statement. In the statement

$$I = 6 * K$$

6 is a constant because it appears in its actual numerical form. (The asterisk indicates the arithmetic operation of multiplication.)

You can write constants in floating point or fixed point form.

Fixed Point Constants

Definition:	A fixed point constant is written **without** a decimal point, using the digits 0, 1, . . . 9. A preceding plus sign or minus sign is optional. The length of the constant cannot exceed 4 digits.
Example:	0 +3 −2496 48

Floating Point Constants

Definition:	Any number written **with** a decimal point, using the digits 0, 1, . . . 9. A preceding plus or minus sign is optional. An unsigned constant is assumed to be positive. The constant may contain an exponent, The exponent, preceded by the letter E, may have a preceding plus or minus sign. All floating point constants are converted to an 8-digit mantissa with a 2-digit exponent. Constants in input data may contain up to 20 digits, but only the first eight significant digits will be carried in the mantissa during calculation.
Example:	42. 1.13 .0046 5000. 6.OE3 (6.0×10^3 or 6000) 6.OE + 3 (6.0×10^3 or 6000) 4264.44 −.00004

Variables

When a quantity in a FORTRAN problem is **not** constant, that is, when its value varies for different executions of the program, or varies at different stages within the program, it is known as a variable quantity. Variable quantities are given names so

they can be identified and referred to by the object program. When reading this description of variables, it is important to distinguish between the **value** of a variable and the **name** of a variable. (When using constants, the **name** and the **value** of the constant are the same.) For example,

VOLT

could be the variable **name** assigned to a **series of values** used in a calculation of current in a circuit. Variables may be in fixed point or floating point.

Fixed Point Variable

Definition:	A fixed point variable name consists of from 1 to 5 alphameric characters (i.e., letters A to Z, digits 0 to 9). The first character **must** be either I, J, K, L, M, or N. The **value** of a fixed point variable cannot exceed 4 digits.
Example:	I JOB 1 MAX N44

The requirement that a fixed point variable must begin with the letters I through N is because these letters have been arbitrarily chosen to indicate to the processor that the values of the variable so named will be in fixed point. Floating point numbers can **never** be the values for a variable defined as fixed point.

Floating Point Variable

Definition:	A floating point variable name consists of from 1 to 5 alphameric characters (i.e., letters A to Z, numbers 0 to 9). The first character in the name must be alphabetic (not numeric) and must **not** be the letters I through N. (Remember, I through N indicate to the processor fixed point values.)
Example:	A B7 DELTA VOLT RATE1

Considerations in Naming Variables

The rules for naming variables allow extensive selectivity. It will be easier for you to follow the flow of a program if you use meaningful symbols wherever possible. For example, to compute distance you could use the statement

X = Y * Z

but it would be more meaningful to write

D = R * T

or even

DIST = RATE * TIME

Similarly, if you want a computation to be performed using fixed point, you could write

I = J * K

or

```
ID = IR * IT
```

or, better yet

```
IDIST = IRATE * ITIME
```

Variables can be written in a meaningful manner by using an initial character to indicate whether the variable is fixed point or floating point and by using succeeding characters as an aid to memory.

Another aid to programming FORTRAN is to vary the last character of a variable name. For example, to compute four different quantities called HRS, you could use the following:

```
HRS 1
HRS 2
HRS 3
HRS 4
```

If the values of these variables were in fixed point, you could precede each of these names by I, J, K, L, M, or N.

The rules for naming and forming variables and constants might be easier to understand if you know how the processor uses the names that you assign. When you establish a name for a constant or variable, the processor establishes for the **object program** a specific location in storage that will contain the data that you have named. Whenever this name appears in the object program, you are, in effect, telling the program to go to the position in core storage where the data, represented by its name, is stored, in order to perform a calculation with the data.

Thus, each constant and variable that you use is assigned a location in 1620 storage where its value is located. Therefore it is important that you remember:

1. When you are forming a constant, do not use more than 4 digits if it is a fixed point number, and be certain to use a decimal if it is a floating point number.

2. When you are naming a variable, use one of the letters I through N as the first character if the value is a fixed point number, and do **not** use the letters I through N if it is a floating point number.

3. Do not assign the same name to more than one variable.

4. Be certain that data is in the same mode (fixed point or floating point) as its variable name indicates it should be.

Subscripts

Variables in your program can be subscripted so that you can represent many quantities with one variable name. In an earlier example, four different quantities called HRS were named HRS 1, HRS 2, HRS 3, and HRS 4. If a program contained 50 quantities for HRS, it would be cumbersome and time consuming to name all of them in this manner.

A group of 50 such quantities can be referred to as an "array." Rather than name all 50 quantities in the array, it is much easier to refer to the entire array by one name and refer to each individual quantity (element) in the array in terms of its place in the array.

For example, assume the following is an array named HRS:

38.6	1st element
40.2	2nd element
36.4	3rd element
.	.
.	.
.	.
47.3	50th element

If you want to refer to the second element in the array, the variable name would be "HRS(2)." The quantity "2" is the subscript to the variable "HRS." (In FORTRAN, language subscripts are always enclosed in parentheses.)

the value of HRS (2) is 40.2
the value of HRS (3) is 36.4
the value of HRS (50) is 47.3

If you want to refer to **any** element of the array, you can write the variable name HRS(I), where I may equal 1, 2, 3, . . . , 50. As you can see by this example, the subscript is **also** a variable. The fact that a subscript can be a variable is extremely important in FORTRAN programming. It means that you can set up a program to do a basic computation, then make the same computation on many different values by merely changing the value of the subscript. This technique is described in a later section.

So far we have only considered arrays that are one dimensional, i.e., there is only one subscript for a variable.

A 1620 FORTRAN program may also use two-dimensional arrays. For example assume the following is an array named MRATE.

	Column 1	Column 2	Column 3
Row 1	14	12	8
Row 2	48	88	4
Row 3	29	25	17
Row 4	1	3	43

If you want to refer to the quantity in row 4, column 2 you would write the variable name MRATE (4, 2).

the value of MRATE (3, 3) is 17
the value of MRATE (1, 2) is 12

If you want to refer to any element of the array, you can write the variable name MRATE (I, J), where I equals (rows) 1, 2, 3, or 4 and J equals (columns) 1, 2, or 3.

Definition:

A subscript can be either a variable or a constant, but must always be positive and in fixed point form.

If *v* represents a variable and *c* represents a constant, then subscripts can be written in the following forms.

v
c
v + *c* or *v* − *c*

Example:

Of subscripts:
IRATE
J
4
NO + 3

Example:

Of variables that are subscripted:
A(J)
K(3)
B(I, J)
I(4,2)
BETA (J—2 ,K+4)

In the last item in the example above, the object program computes the value of the two-dimensional subscript by subtracting 2 from the value of J and adding 4 to the value of K.

Arithmetic Statements

The numerical calculations to be performed in the object program are defined by arithmetic statements. FORTRAN arithmetic statements closely resemble conventional arithmetic formulas. They contain a variable to be computed, followed by an = sign, followed by an arithmetic expression. For example, the arithmetic statement

$$Y = A - SIN(B)$$

means "replace the value of the variable on the left side of the equal sign with the value of the expression on the right side of the equal sign." In a FORTRAN program, the equal sign means "is to be replaced by" rather than "is equivalent to."

The meaning of the equal sign is important in FORTRAN. Earlier in the manual we learned that each variable in the object program is assigned a specific location in storage that contains the data you have named. As an example, assume a fixed point variable named NUMBR has the value of 6. The statement

$$NUMBR = NUMBR + 2$$

would cause the object program to take the value of NUMBR, which is 6, increase it by 2, and then set the result 8 as the new value of NUMBR.

Format:

"a = b"
a is a variable and may be subscripted
b is an arithmetic expression (explained later)

Example:

A = B + C
D(I) = E(I) + 2.—F

Expressions

An expression in FORTRAN consists of a series of constants, variables, and functions (explained later) separated by parentheses, commas, and/or operation symbols, so as to form a mathematical expression. Expressions appear on the right-hand side of arithmetic statements.

Operation Symbols

Five basic operations can be used in FORTRAN: addition, subtraction, multiplication, division, and involution (raising to a power). These operations are represented in FORTRAN by the following symbols:

+	addition
—	subtraction
*	multiplication
/	division
**	involution

Rules for Forming Expressions

There are five rules that you must follow when you write FORTRAN arithmetic statements. The purpose of these rules is to help you write your statement correctly in FORTRAN language.

1. The constants and variables used in a FORTRAN expression may be either in fixed point or floating point mode, but both modes cannot be used in the same expression. For example:

426	Constant – fixed point mode
3.	Constant – floating point mode
I	Variable – fixed point mode
R	Variable – floating point mode
HRS (J)	Subscripted variable – floating point mode

   In the last example, the subscript J, used with the floating point variable HRS, is in fixed point mode. The mode of the expression is determined only by the mode of the quantity. Using a fixed point subscript with a floating point variable does not violate the rule of mixing modes in an expression.

2. Involution of a quantity does not affect the mode of the quantity. However, a fixed point quantity may never be given an exponent. The following are valid.

```
A**B   floating point
A**J   floating point
```

3. Whenever two operation symbols follow in succession, they must be separated by parentheses. The following examples illustrate this rule:

Mathematical Expression	FORTRAN Expression	Incorrect FORTRAN Expression
$\frac{A}{-B}$	A/(—B)	A/—B
AB or A . B	A*B	AB
$A^{E+2.}$	A**(E + 2.)	A**E + 2.
$A^{E+2.}B$	A**(E + 2.) *B	A**E + 2. *B

   Common algebraic rules must also be observed. For example the ambiguous mathematical expression

$$R^{A^{C}}$$

   can be written as R**(A**C) or as (R**A) **C, whichever it is intended to be.

   The mathematical expression

$$\frac{AB}{CD}$$

   can be correctly written as A*B/(C*D) or as A/C * B/D. But the expression A*B/C*D, although it is a valid FORTRAN expression, does not represent the mathematical expression $\frac{AB}{CD}$.

4. Parentheses are used to specify the order of operations in an expression. If parentheses are omitted, the order is taken to be from left to right as follows:

**	involution (raising to a power)
* and /	multiplication and division
+ and —	addition and subtraction

For example the FORTRAN expression

$$A + B/C + D **E*F - G$$

will be taken to mean the mathematical expression

$$A + \frac{B}{C} + (D^{E} \cdot F) - G$$

The FORTRAN expression could have been written with parentheses as follows:

$$A + (B/C) + (D**E*F) - G$$

5. A sequence of consecutive multiplications and divisions (or consecutive additions and subtractions) without parentheses will be grouped from the left. For example:

$$A * B * C * D * E$$

will be taken to mean

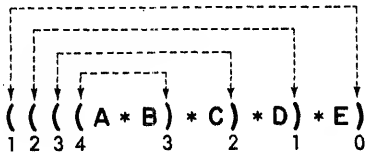
$$((((A*B)*C)*D)*E)$$

Until you become proficient in writing FORTRAN programs, always use parentheses to specify the order of operations.

Verification of Correct Use of Parentheses

To check a complicated FORTRAN expression to determine that the parentheses are correctly inserted in pairs, use the following method.

Working from the left to right, label the first open parenthesis "1," and increase the label by 1 for each open parenthesis and decrease it by 1 for each closed parenthesis. The label of the last parenthesis should be 0. The mate of an open parenthesis labelled *n* will be the next parenthesis labeled *n-1*.



Mode of an Arithmetic Statement

The difference between a FORTRAN **expression** and a FORTRAN **arithmetic statement** should be emphasized at this time. By definition, an arithmetic statement is composed of a single variable on the left and an arithmetic expression on the right. This distinction is important because, although an expression must not be in mixed mode (containing both fixed point and floating point quantities), an arithmetic statement may be in mixed mode. For example, when you write

$$A = (I*J)/K$$

(I*J)/K is an expression and A = (I*J)/K is an arithmetic statement.

If an arithmetic statement is in mixed mode, the mode of the variable on the left side of the equal sign determines the mode of the result.

If the variable on the left is in fixed point and the expression on the right is in floating point, the expression will first be evaluated in floating point, the portion following the decimal point will be dropped, and the remainder will be converted to fixed point by retaining only the four digits immediately to the left of the decimal. If a result is

123456.78

the fixed point quantity stored is 3456.

If the variable on the left is in floating point and the expression on the right is in fixed point, the expression will be evaluated in fixed point and the result converted to floating point. For example,

Arithmetic Statement	Result
A = 5/3	A = 1.
A = 5./3.	A = 1.6666666
I = 5/3	I = 1
I = 3./2.	I = 1
I = 123456.78/4.	I = 864 (was computed as 30864.195)

FORTRAN Arithmetic

If your problems are programmed in floating point rather than in fixed point, you will find it is easier to process fractions because you will not have the problem of locating decimal points. If a particular problem that you are programming requires the use of fixed point quantities, you must understand exactly how fixed point arithmetic is accomplished.

In fixed point calculations, if the result is not an integer (whole number) the result is **truncated** to a whole number. That is, the fractional portion of the result is discarded, and no rounding takes place.

The fixed point division 5/3 is 1, not 2. Therefore, if you write an expression with a series of operations that includes a division, you must be careful when grouping. For example,

```
A = 5./3. * 4.
```

In floating point, 5 divided by 3 equals 1.6666666, and this value multiplied by 4 equals 6.6666664.

If this arithmetic statement is written in fixed point,

```
I = 5/3 * 4
```

then 5 divided by 3 equals 1.6, which is truncated to 1. The 1 is multiplied by 4 and the answer is 4.

If you had reversed the grouping in the statement,

```
I = 4 *5/3
```

the result would be 6. Remember, in a statement with a series of multiplications and divisions where the parentheses have been omitted the operations are performed from left to right.

Functions

Assume that you are writing a FORTRAN program that requires taking the square root of a number at different locations in the program. The statements to perform the square root would be identical, except for the different arguments used each time.

Instead of writing the same statements many times, the FORTRAN program allows you to take the square root of a number by merely inserting the expression "SQRT (x)" into an arithmetic statement wherever it is required. The mathematical operations which are required to take the square root of a number are "prewritten" into the FORTRAN program as a **subroutine.** (A subroutine is a program which performs certain operations and may be included in another program to cause those operations to be carried out each time the subroutine is used.)

The following functions can be used in FORTRAN:

Mathematical Function	FORTRAN Name
Square Root	SQRT
Exponential	EXP
Sine of an angle in radians	SIN
Cosine of an angle in radians	COS
Arctangent, angle given in radians	ATAN
Natural logarithm	LOG

For each of the functions shown above, there exists a subroutine within the FORTRAN system which computes the function of the argument enclosed in the parentheses. These subroutines will be compiled into the object program automatically when called for by a statement containing the name of one of these functions. (These subroutines are sometimes called "relocatable subroutines").

To take the square root of a quantity with the variable name DELTA, you could write the statement

```
D = SQRT (DELTA)
```

The argument enclosed in the parentheses must follow the name of the function. The argument can be a variable or an expression and the variable can be subscripted. The argument must always be in floating point mode. For example:

```
A = COS (B7)
A = SQRT (BETA)
A = LOG (A)
Y = A — SIN (B * SQRT (A))
```

Control Statements

FORTRAN statements are executed in the object program in the same sequence as they are written on the coding sheet, **unless you specify a different sequence.**

Control statements provide flexibility in program development. If statements could only be followed sequentially in a fixed pattern, a program would follow a single path of operation without any possibility of dealing with predefined exceptions to the procedure, and without any ability to choose alternatives based upon conditions encountered during the processing of the program.

As an example of the program control that can be exercised, assume that you have written a FORTRAN program consisting of fifteen statements. These statements perform a number of operations upon a series of variable quantities. Now, if the first ten statements develop meaningless results when processed with variable quantities of zero, the processing time of the object program would be reduced if the first ten statements could be bypassed when the quantity to be processed is zero. A single FORTRAN control statement permits you to evaluate a quantity, and depending upon the value, permits you to direct the program to some other statement rather than have the program continue in the sequence of the statements.

In a previous section of this part of the manual, it was stated that you can set up a program to perform a basic computation, then make the same computation on many different values simply by changing the value of the subscripts. This kind of operation is called "looping." You would use a control statement to establish the number of statements that are to be included in the basic computation to determine how many times the loop is to be performed.

Control statements that direct the program to another statement in the program are referred to in this manual as program transfer statements.

Statement Numbers

Whenever you direct the program to a specific statement (other than the next sequential statement), the statement must be numbered so that it can be identified by the processor. On the FORTRAN coding form, the first five columns are reserved for statement numbers. A statement number can be any number from 0001 to 9999 (leading zeros are not required).

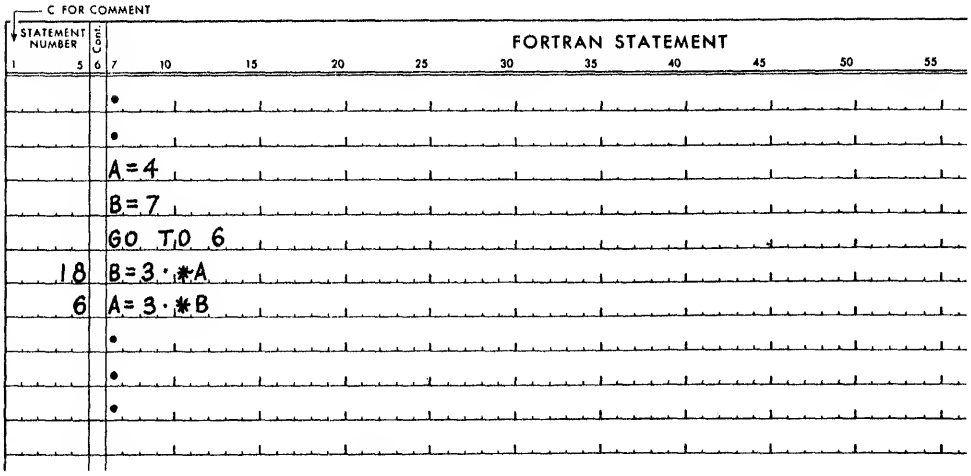
Statements must be numbered only when they are referenced by another statement and no two statements can have the same number. Also, there is no requirement that every statement must have a number, nor that statements must be numbered in sequence. It is possible to number every statement as an aid in programming, but each number you assign requires positions of storage. If the problem being programmed is very long and requires a large amount of storage, you may not be able to afford the luxury of numbering every statement.

Unconditional GO TO

This statement interrupts the sequential execution of statements, and specifies the number of the next statement to be performed.

Format:	"GO TO *n*" where *n* is a statement number.
Example:	GO TO 30 GO TO 1000

A coding example is shown below:



The GO TO statement transfers the program to statement 6 where the result 21 is obtained.

Computed GO TO

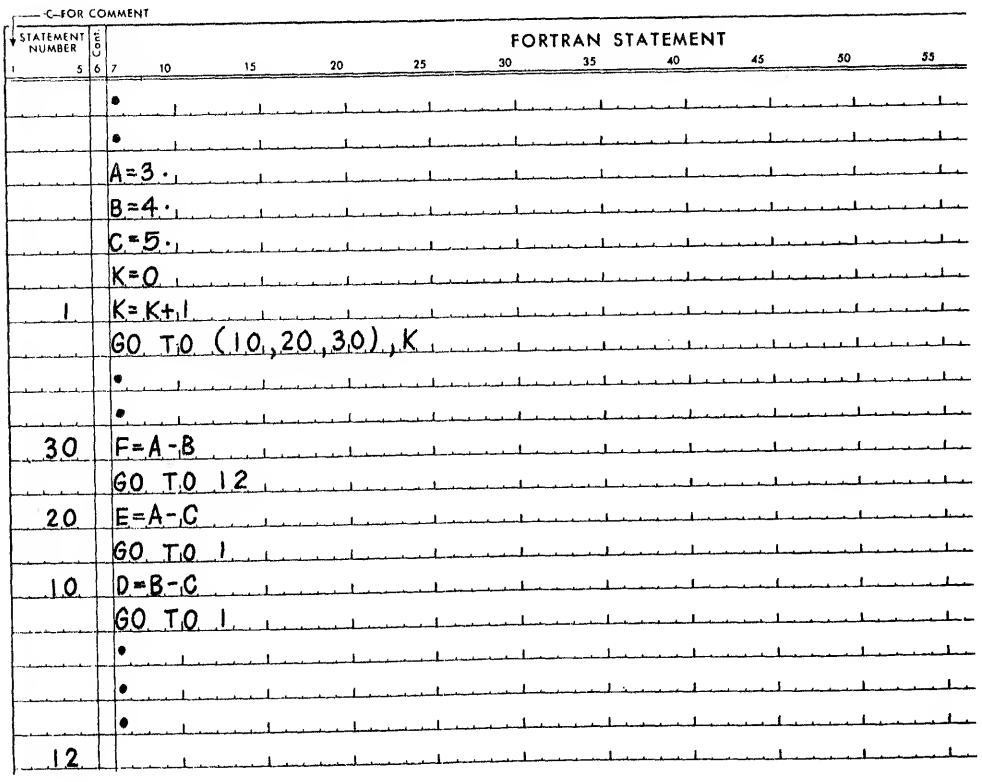
This statement also specifies the number of the next statement to be performed. It is different from the unconditional GO TO, because the statement number that the program is transferred to can be altered **during** the program in a computed GO TO statement.

Format:	"GO TO (n_1, n_2, . . ., n_m), i" where n_1, n_2, . . ., n_m are statement numbers and i is a fixed point variable. The variable cannot be subscripted. The parentheses enclosing the statement numbers, the commas separating the statement numbers, and the comma following the right parenthesis, are all required punctuation.
Example:	GO TO (3, 4, 5), L GO TO (4, 4, 8, 14, 24), M

The computed GO TO statement transfers the program to the 1st, 2nd, etc., statement number in the list depending upon whether the value of i is 1, 2 . . ., etc. The variable i must never have a value greater than the number of items on the list in the parentheses.

In the first example above, if the value of L is 2, the program is transferred to statement 4. In the second example, if the value of M becomes 1 or 2 the program is transferred to statement 4. If it becomes 3, 4, or 5, the program is transferred to statements 8, 14, and 24, respectively.

A coding example is shown below:



In the example, D, E, and F are computed, in that order, and the program is transferred to statement 12. This is a simplified example used to illustrate a computed GO TO statement. If these were the only computations in a program, you would probably just compute D, E, and F in sequence without using a computed GO TO statement.

IF Statement

The IF statement permits you to transfer the program to a particular statement depending upon the value of an expression.

Format: "IF *(a)* n_1, n_2, n_3"
where *a* is an expression and n_1 n_2 and n_3 are statement numbers.

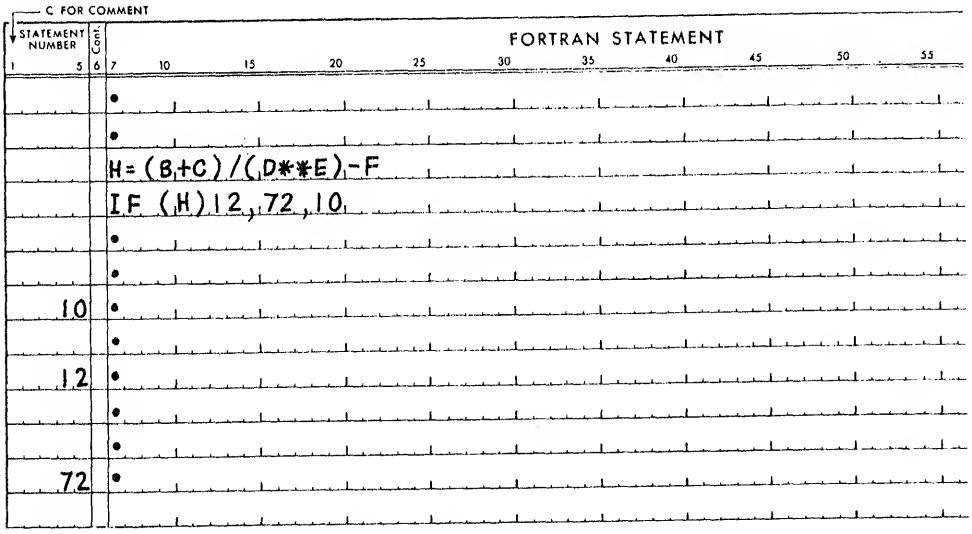
The expression must be enclosed in parentheses. The statement numbers must be separated from each other by commas.

Example:
```
IF (A — B) 10, 5, 7
IF (A(I)/D) 1, 2, 3
```

The program is transferred to statement number n_1, n_2, n_3, depending upon whether the value of the expression *a* is less than, equal to, or greater than zero, respectively.

In the first example, if both A and B have the value of 2, the program is transferred to statement number 5. In the second example, if the result of the expression is greater than zero, the program is transferred to statement number 3.

Suppose a value, HRS, is being computed. Whenever this value is positive, the main routine of the program should be followed. Whenever the value of HRS is negative, an alternative routine starting at statement 12 is to be followed, and if HRS is zero, an error routine at statement 72 is to be followed. This problem can be coded as follows:



IF (SENSE SWITCH) Statement

This statement permits the program to transfer to a particular statement depending upon the setting of any one of the four console program switches.

Format: "IF (SENSE SWITCH *i*) n_1, n_2"
where *i* is the number of one of the console program switches, and n_1 and n_2 are statement numbers.

The parentheses, enclosing the words SENSE SWITCH, and the commas, separating the statement numbers, are required punctuation.

Example:
```
IF (SENSE SWITCH 3) 14, 50
IF (SENSE SWITCH 1) 20, 40
```

The program transfers to the statement numbered n_1 when the designated program switch is on, or to the statement numbered n_2 when it is off.

PAUSE Statement

The PAUSE statement is used as a convenient means of causing the object program to halt temporarily. Halting the object program is sometimes required so that the machine operator may check part of the output to determine if one or more values are within predetermined limits before continuing with the program. The PAUSE statement is also useful as an aid in the initial testing of a new program. PAUSE statements, located at the end of one or more phases in a program, permit you to check the accuracy or validity of a part of a problem by checking the data obtained in that part before altering the data in subsequent operations in the program.

Format:	"PAUSE"
Example:	PAUSE

The PAUSE causes the computer to halt. Pressing the start switch causes the program to resume with the statement following the PAUSE statement.

STOP Statement

This statement causes the computer to halt during the processing of the object program, to return the typewriter carriage, and to type the word "stop." In contrast to the PAUSE, this statement is used where a final, rather than a temporary, stop is required.

Format:	"STOP"
Example:	STOP

DO Statement

As discussed earlier, the ability of the FORTRAN program to repeat the same operations with different data, called looping, is a powerful tool which greatly reduces programming effort. There are several ways to accomplish looping, one way is to use an IF statement. For example, assume that a plant carries 1,000 parts in inventory. Periodically it is necessary to compute stock on hand of each item (INV), by subtracting stock withdrawals of that item (IOUT) from a previous stock on hand.

It would be wasted effort to write a program which would indicate each of the 1,000 separate subtractions by a separate statement. (It would also waste computer storage, since each separate instruction to the computer must be in computer storage.) The same results could be achieved by the following program:

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–55)
	•	
	•	
5		J=0
10		J=J+1
25		INV(J)=INV(J)-IOUT(J)
15		IF(1000-J)20,20,10
20	•	
	•	
	•	
	•	

An index, J, is established which will be increased by 1 each time statement 10 is executed. Statement 5 sets J to zero (this statement is processed on the first loop only) so that statement 10 will set J equal to 1 for the first execution of statement 25.

Statement 25 will compute the current stock on hand by subracting the stock withdrawal from the previous stock on hand. The first time statement 25 is executed, the stock on hand of the first item in inventory, INV(1), will be computed by subtracting the stock withdrawal of that item, IOUT(1). Statement 15 tests to determine if all items in stock have been updated. If not, the expression 1000-J will be positive and the program will transfer to statement 10, which will increase the value of J by 1. Statement 25 will be executed again, this time for the stock on hand of item 2, INV(2), and the stock withdrawal of item 2, IOUT(2). This procedure will be repeated until the stock of item number 1000 has been updated. At this point, J will be equal to 1000, and the expression in statement 15 will be equal to zero. At this time, statement 15 will cause the program to transfer to statement 20 in order to continue with other parts of the program.

Notice that three statements (5, 10, and 15) were required for this looping which could have been accomplished with a single DO statement.

The purpose of the DO statement is to simplify the programming of loops and to provide greater flexibility in looping.

Format:

"DO n $i = m_1, m_2, m_3$"
where n is a statement number, i a fixed point variable, and m_1, m_2 and m_3 can be either a fixed point constant or a fixed point variable.

Subscripts and sign indication are not permitted in a DO statement.

If m_3 is not stated, it is taken to be 1.

The commas are required punctuation.

Example:

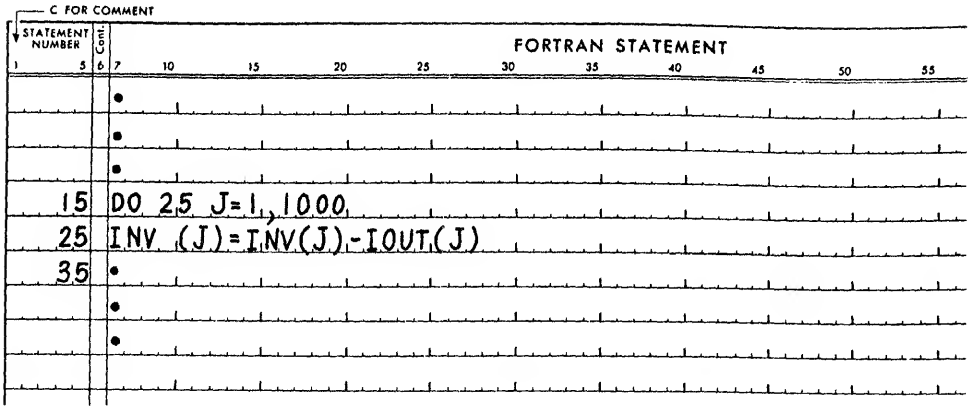
```
DO 20 JBNO = 1, 10
DO 20 JBNO = 1, 10, 2
DO 20 JBNO = K, L, 3
DO 16 K = 1, M
DO 16 J = L, 2
DO 18 INDEX = J, K
```

The DO statement is a command to repeatedly execute the statements that follow the DO statement, up to and including the statement with the number equal to the value of n. The first time through the loop, the statements are executed with i equal to the value of m_1. For each succeeding execution of the statements, i is increased by the value of m_3. After the statements have been executed with i equal to the highest value that does not exceed m_2, the program transfers to the statement which follows the last statement in the range of the DO (the statement after statement number n).

Thus, the DO statement does three things:

1. It establishes an index which may be used as a subscript or in a computation.
2. It causes looping through any desired series of statements as many times as required.
3. It increases the index (by any amount specified) for each separate execution of the series of statements in the loop.

In the example below, an inventory problem is programmed using the DO statement.



Statement 15 is a command to execute the following statements up to and including statement 25; the first time the value of J will be 1, thereafter the value of J will be increased by 1 for each execution of the loop until the loop has been executed with the value of J equal to 1000. After the loop has been executed with J equal to 1000, the statement following statement 25 will be executed.

The following is a comparison of statement 15 with the format of a DO statement, and an introduction to some of the terms used in describing a DO statement.

DO Format	n	i	=	m_1,	m_2,	m_3
DO Statement	25	J	=	1,	1000	
	Range	Index		Initial Value	Test Value	Increment

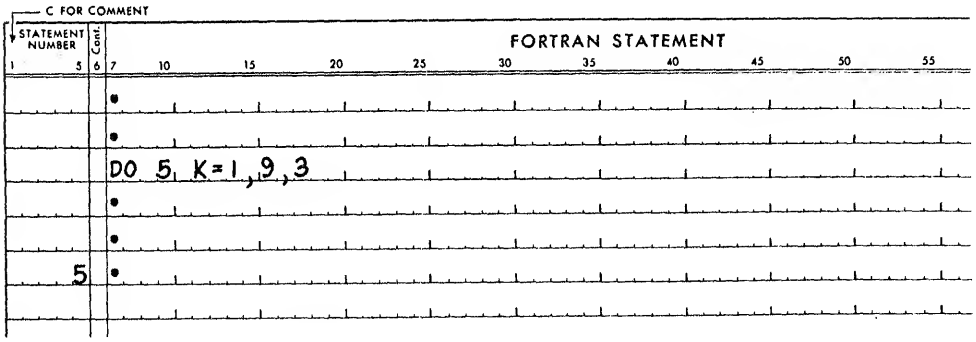
The **range** is the series of statements to be executed repeatedly. It consists of all statements following the DO statement up to and including statement n. In this case, statement n is statement 25, and the range consists of only one statement. The range can consist of any number of statements. (NOTE: throughout the remainder of the manual, the word DO means the DO statement and all statements within the range of the DO statement.)

The **index** is the variable which will change for each execution of the range. In the example, the index J was also used as the subscript to the variables in statement 25. Thus, it served two purposes: to maintain a count of the number of loops executed, and to establish the correct variable for each execution of the loop.

The **initial value** is the value of the index for the first execution of the range. Although the initial value was 1 for this example, in another problem it might be some other quantity. Often, the initial value will change at different times within the program. In such cases it may be stated as a fixed point variable rather than as a constant, as in the example. If it is a variable, its value must be set up in a statement that precedes the DO statement.

The **increment** is the amount by which the value of the index will be increased after each execution of the range. In the example, it is not coded because the increment desired is 1 and the DO statement automatically uses 1, unless some other value is specified. As with the **initial** value, the increment may be written as a fixed point variable.

The **test value** is the value which the index may not exceed. After the range has been executed with the highest value of the index which does not exceed the test value, the DO is satisfied and the program continues with the first statement following the range. In the example, the DO was satisfied after the range was executed with the index value equal to the test value. In some cases, the DO is satisfied **before** the test value is reached. Consider the following:

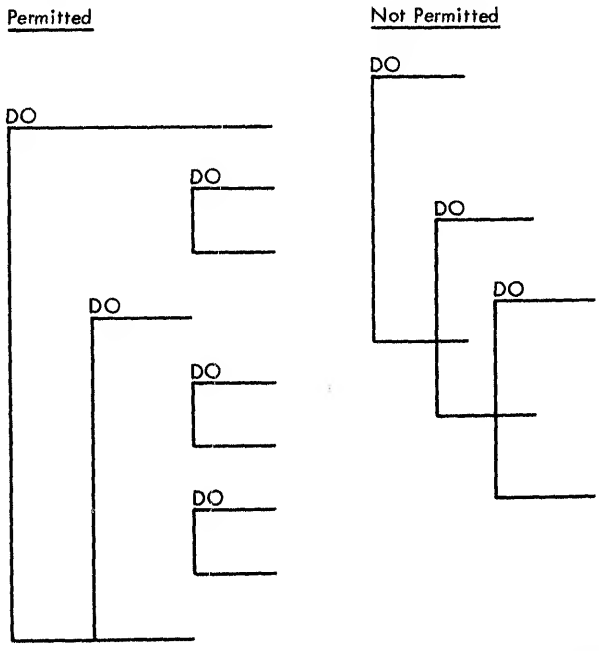


In this example, the range will be executed with K equal to 1, 4, and 7. The next value of K would be 10, but since this exceeds the test value, the program transfers to the statement following statement 5 after the range is executed with K equal to 7. Note that after the transfer, the **index value** K (10) was **not** the same as the **test value** (9).

DO Statements Located Within a DO Statement

One or more DO statements may be included within the range of a DO statement. When this is done the following rule must be observed:

> If the range of a DO statement (the **major** DO statement) includes another DO statement (the **minor** DO statement), all statements in the range of the minor DO must also be in the range of the major DO.
>
> This rule is illustrated in the drawing below. (Brackets are used to illustrate the range of a DO.)



In 1620 FORTRAN it is possible to transfer into the range of a DO statement from outside its range by either an IF or GO TO statement. (This cannot be done on many other FORTRAN programming systems, and, if there is a possibility that the source program will be compiled on some other FORTRAN system, you should not use this technique.)

If you write a statement to transfer into the range of a DO from outside of its range, you must understand that the value of the index is not reset by such a transfer. You may use the current value of the index or you may redefine the index prior to the transfer. If you want to use the current value of the index, read the next paragraph carefully, and then review the explanation of "Test Value." The current value of the index in a problem may not be what you think it is.

Preservation of Index Values

When the program transfers out of the range of a DO in the normal manner (that is, when the DO becomes satisfied and the program transfers to the next statement after the range), the exit is defined as a **normal exit.** After a normal exit from a DO occurs, the value of the index is **not** redefined to its original value. To determine the value of the index after a normal exit, remember that after a loop is completed, the index is increased by the increment **before** it is tested to determine if additional loops are to be taken.

When the program transfers out of the range of a DO by an IF or GO TO statement, the value of the index is its current value at the time of the transfer.

In both types of exits, the current value of the index is preserved for any subsequent use. If the exit occurs by a transfer which is in the range of several DO's, the current values of all the indexes controlled by those DO's are preserved for any subsequent use.

Restriction on Statements Used In the Range of a DO

The range of a DO cannot contain any statement which redefines the value of the index or the value of any of the indexing parameters (m_1, m_2, *or* m_3). The indexing of a DO statement must be completely set before the range of the DO is entered.

The first statement in the range of a DO **cannot** be a nonexecutable statement. A nonexecutable statement is a statement in the source program that does not create instructions in the object program. The control statement CONTINUE and the specification statements DIMENSION and FORMAT are the only nonexecutable instructions. These instructions are described later.

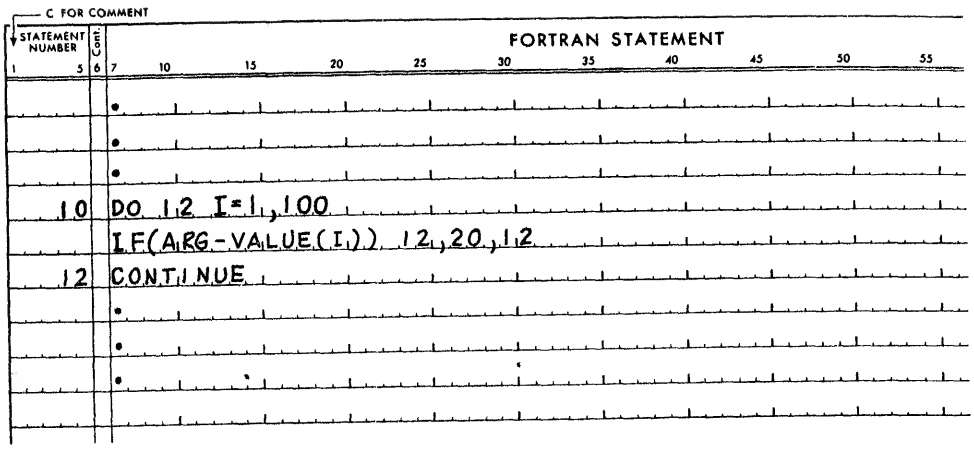
The last statement in the range of a DO must not be a program transfer statement (IF or GO TO, etc.)

CONTINUE Statement

This statement is used as the last statement in the range of a DO when the last statement would otherwise be a program transfer statement (see rule previously given.) This statement does not create any instructions in the object program.

Format:	"CONTINUE"
Example:	CONTINUE

Consider the following table search program which requires a CONTINUE statement. This program will scan the 100-entry array named VALUE until it finds an element which equals the value of the variable named ARG, then the program will transfer to statement 20 with the value of I available for use. If no element in the array is equal to the value of ARG, the program is transferred to statement 12. No operations are performed by the CONTINUE statement; the program merely continues with the next sequential statement following statement 12.



END Statement

The END statement is a signal to the compiler that the end of the source program has been reached.

Format:	"END"
Example:	END

The object program will not be compiled unless the END statement appears as the last statement in the source program.

Some Thoughts About Programming FORTRAN

Learning how to program FORTRAN can be divided into two phases. One might be called "How to write statements that perform calculations upon data." The second could be called "How to get data into and out of the program."

After you have programmed a few problems in FORTRAN, you will find that most of your programming time will be concerned with calculations upon data; moving data into and out of the program will be of secondary importance. However, getting data into and out of the program may be the most difficult part of FORTRAN to learn because it may involve concepts with which you are not familiar.

A brief review at this time should help.

1. The 1620 system consists of a Central Processing Unit with a typewriter for entering or printing out data. The system may contain a 1621 Paper Tape Reader, a 1624 Paper Tape Punch, or 1622 Card Read Punch.
2. The IBM FORTRAN processor may be punched in either IBM cards or paper tape, depending upon the type of 1620 system you have.
3. The FORTRAN processor is read into the 1620 first, followed by paper tape or card records containing the source program. The result of this compilation is an object program containing 1620 machine language instructions.
4. The object program (in cards or paper tape) is then placed into the 1620, followed by card or tape records containing the data that is to be processed.
5. The results of the computations are either printed on the typewriter, punched into cards, or punched into paper tape.

The remainder of this part of the manual, WRITING THE FORTRAN PROGRAM, is concerned with statements that move data into and out of the program, statements that determine how much data will be read into the program, and the kind of data that is read (fixed point or floating point).

Input/Output Statements

Input statements are used to read data into the program and output statements are used to print or punch the results of calculations.

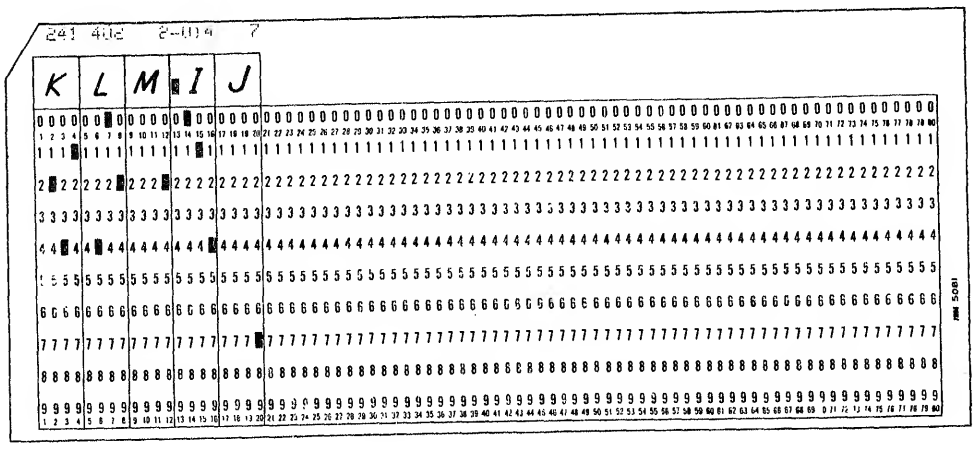
Consider the following mathematical problem

$$Y = \frac{I}{K} \cdot \frac{J}{L} + M \cdot N$$

and assume that I, J, K, L, and M are variable quantities punched into a file of cards, with N equal to 48. If you write the source statement

$$N = 48$$

the processor sets aside an area in storage (in the object program) called N, and sets up the necessary instructions to place the quantity 48 in that area at object time. The variables may be punched into the cards as shown below.



Remember that the data cards are not available while the source program is being compiled. Therefore, in this example, an input statement would be included in the source program to tell the processor that the object program will read the input data from cards (that there will be five data fields in the sequence of K, L, M, I, and J). The input statement also contains the statement number of another type of statement called FORMAT. In this example, the FORMAT statement, referenced by the input statement, tells the processor that each field named in the list is four columns long and that the quantities are in fixed point mode. Thus, as the object program is executed, each data card is read by the program and the quantity fields are placed into the storage locations named K, L, M, I, and J. When processing is completed for the set of values in one card, the values for the next card are read into the same storage locations and processing for the new set of values is performed.

Thus, an input or output statement, each with its corresponding FORMAT statement specifies the number and sequence of the data input or output fields, the length of the data fields, and whether they are in fixed or floating point mode.

NOTE: FORMAT statements are described in a later section entitled SPECIFICATION STATEMENTS.

READ Statement (Card Input)

This statement causes data to be read from a card in the 1622 Card Read Punch.

Format:	"READ *n*, *List*" where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be transmitted.
Example:	READ 4, A, B, C READ 30, HRS1, HRS2, HRS3 READ 2, VOLT(I), OHM(J)

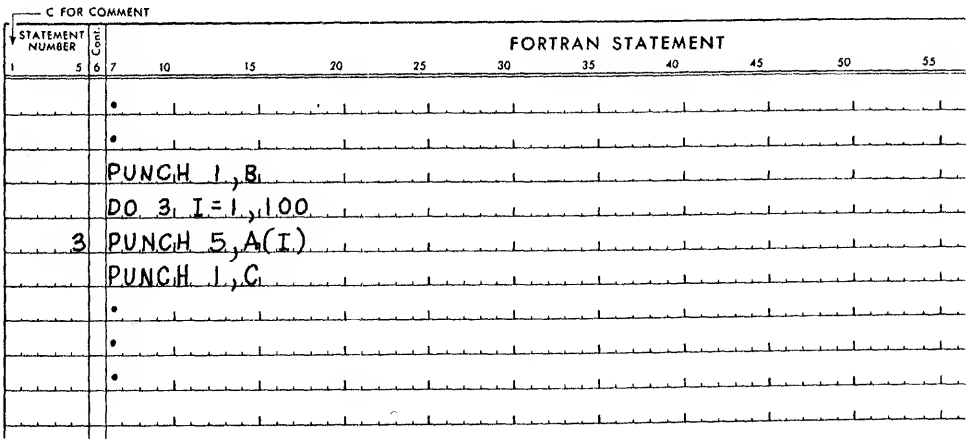
The READ statement causes data to be read from a card (at object time) and causes the quantities from the card to become the values of the variables named in the list.

If the quantities for a set of values require more than 80 digits (the number of columns in an IBM card) the program reads successive cards until the complete list of quantities has been "satisfied," i.e., the data for all of the variables has been brought in and stored in the locations specified by the READ statement.

If there are more quantities in the card than there are on the list, only the number of quantities specified on the list are read by the program. Thus, if a card contains five quantities, but the list in the READ statement contains only two, the remaining three quantities are ignored.

It is important to note that every item in a list corresponds to one and only one quantity. Arrays whose members are so numerous that itemizing them in a list is impractical may be handled by using a shorter list and including the input or output statement in the range of a DO.

For example, suppose items B, A, and C are to be punched, in that order, and A represents a one-dimensional array consisting of 100 elements. The output statements could be written in the following manner:



The DO would punch the 100 elements of the array A.

ACCEPT Statement (Typewriter Input)

This statement is used when input data is to be entered into the program from the console typewriter.

Format:	"ACCEPT *n*, *List*" where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be typed.
Example:	ACCEPT 30, A, B, C, D(3)

This statement selects the typewriter as the input device, returns the typewriter carriage, and halts the computer to await manual entry of data. Data must be typed in accordance with the FORMAT statement until the complete list is satisfied.

ACCEPT TAPE Statement (Paper Tape Input)

This statement is used when input data is to be entered into the program from the 1621 Paper Tape Reader.

Format:	"ACCEPT TAPE *n*, *List*" where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be entered.
Example:	ACCEPT TAPE 48, K, A(J)

Paper tape records are read into storage until the complete list is satisfied.

PUNCH Statement (Card Output)

Format:	"PUNCH *n*, *List*" where *n* is the statement number of a FORMAT statement, and *List* is a list of the quantities to be punched.
Example:	PUNCH 1, A, D, C PUNCH 2045, SQRT

One or more cards are punched until the complete list has been satisfied.

TYPE Statement (Typewriter Output)

Format:	"TYPE *n*, *List*" "PRINT *n*, *List*" where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be typed. The words TYPE and PRINT are interchangeable.
Example:	TYPE 19, X, Y PRINT 2, DELTX

One or more lines are typed until the complete list is satisfied.

PUNCH TAPE Statements (Paper Tape Output)

Format:	"PUNCH TAPE *n*, *List*" where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be punched.
Example:	PUNCH TAPE 4, A, B, C PUNCH TAPE 100, AVGHR

One or more records are punched until the complete list is satisfied.

Specification Statements

There are two types of specification statements, FORMAT and DIMENSION. The FORMAT statement has already been defined as a statement that tells the FORTRAN processor the length of each input or output data field, and whether the field is (or will be) in fixed point or floating point mode. The DIMENSION statement provides the processor with the information necessary to allocate storage in the object program for arrays of quantities.

A DIMENSION statement does not create instructions in the object program. Its function is merely to supply information to the processor.

FORMAT Statement

The FORMAT statement permits you to determine how you want the results of computations to look in the output data and allows you to tell the processor how to punch or type the input data. In both instances, you are concerned with the problem of converting data from either its external form (cards, tape, typewriter) to an internal form (core storage), or from its internal form to cards, paper tape, or the printed line.

Format:	"FORMAT (s_1, s_2, s_3, . . ., s_n)" where s_1, s_2, s_3 and s_n are specifications as described below.
Example:	4 FORMAT (I2/F10.4, E12.4) 6 FORMAT (I2, I4) 3 FORMAT (E12.4, I5)

FORMAT specifications have three forms:

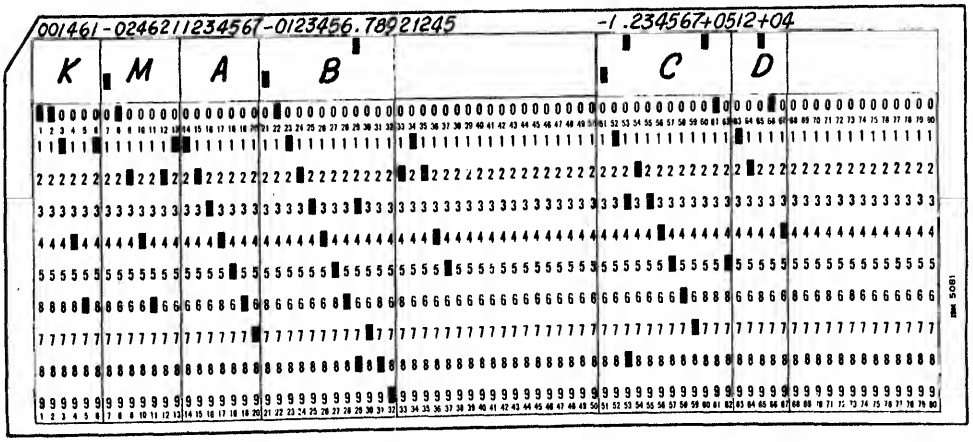
Type	Format	Description
I	Iw	Fixed point numbers
F	F$w.d$	Floating point numbers without an exponent
E	E$w.d$	Floating point numbers with an exponent

where w is the width of the field (that is, the total number of positions printed or punched) and d is the number of decimal places as explained in the following paragraphs.

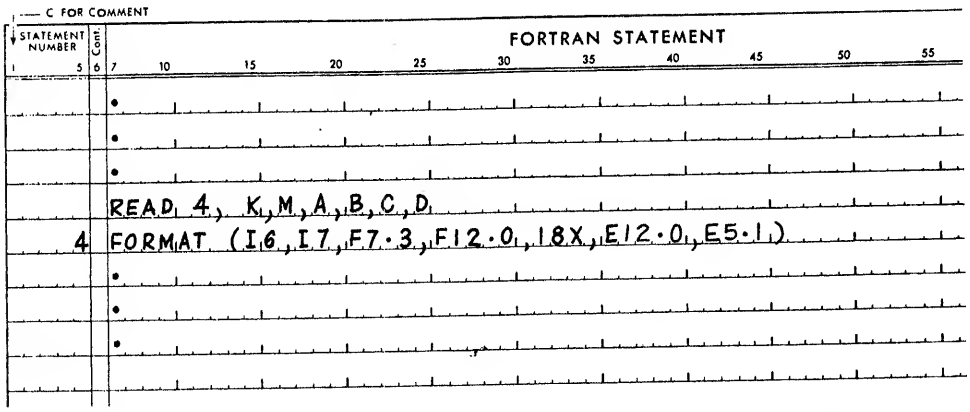
All three forms can be used in FORMAT statements for both input and output statements. However, there is a slight difference in the meaning between an input and an output specification, so they will be described separately. The FORMAT statement may be written anywhere in the source program (except as the first instruction of a DO).

Input Specifications

An example of card input is used here, but the principles illustrated also apply to paper tape records and to typed input.



The statements required in the source program to enable the object program to read this data card are:



The input card in this example contains a variety of formats, and, although it is not a representative example of an input card that might be used in an actual program, it does show how various data formats can be read into the program. The object program will store these values as shown in the chart and text that follow.

Item No.	Variable Name	Punched In Card As:	FORMAT Specification	Considered By The Object Program To Have The Value Of:	Stored In The Object Program As:
1	K	001461	I6	+1461	000000 1461
2	M	-024621	I7	-4621	000000 4621
3	A	1234567	F7.3	+1234.567	M E 1234567 04
4	B	-0123456.789	F12.0	-123456.78	M E 1234567 8 06
5	-	21245 (and 13 blanks)	18X	(No value taken; these columns skipped)	
6	C	-1.234567+05	E12.0	-123456.7	M E 1234567 0 06
7	D	12 +04	E5.1	12000.	M E 1200000 05

1. The first variable in the card, K, has been assigned the FORMAT of I6. This format tells the processor that the first field in the card contains 6 columns and that the number is in fixed point mode. The maximum size of a fixed point number is 4, so at object time the leading zeros are truncated and the value is stored as +1461. The value is assumed to be plus because no sign indication is given.

2. The next variable, M, has been assigned the FORMAT of I7. This format tells the processor that the next field in the card consists of 7 columns and that the number is in fixed point mode. The value will be stored as —4621 with the high-order digits truncated. A number would be truncated in this manner only if you **wanted** to lose the high-order digits, or if you happened to set the specification in error. It is important to note that the width *(w)* must be the width of the field on the card, not just the number

of positions of the card that you want to bring into the program. In this example, if the specification of K had been I4, the processor would assume that the value for K was located in columns 1 to 4, that the value for M was punched beginning with column 5, and that the value of A was punched beginning with column 12, etc.; thus all subsequent fields would be read incorrectly.

3. The floating point variable A has FORMAT F7.3. This format tells the processor that the value is a floating point number, that the field in the card contains 7 columns, and that there are 3 digits to the right of the decimal place. The object program will consider A to have the value of 1234.567, then "places" the decimal point to the left of the high-order (leftmost) digit, and sets the exponent as 04 for this value to account for the number of positions that the decimal point was shifted.

4. The floating point variable B has FORMAT F12.0. This format tells the processor that the value is a floating point number, and that the field in the card contains 12 columns. Because the decimal point is punched in the card in its proper place, a specification for *d* is not required, and if specified in error would be ignored by the object program. The maximum size of a mantissa in floating point is 8 positions, so the low-order positions of the quantity are truncated. When the value is stored, the decimal point is adjusted 6 places and the exponent is set to 06.

5. Columns 33 through 50 contain blank columns and punched data that are not required in the program. These columns are ignored by giving the processor the specification 18X. This specification reads these columns into storage as blank characters. The maximum number of columns that can be ignored with one specification is 49.

6. The floating point variable C is punched into the card using a standard mathematical notation: that is, the decimal point has been adjusted to a specific location and the magnitude of the quantity is given by punching the exponent in the columns following the value. The FORMAT for C, E12.0, tells the processor that the value is in floating point mode with an exponent to indicate its magnitude, and that the field in the card contains 12 columns. Because the decimal point has already been punched into the card, the specification *d* is ignored. The object program will assume that C has a value of −123456.7 by noting where the decimal is punched in the field and what the value of the exponent is. When the value is stored, the decimal is adjusted to the left of the high-order digit, and the exponent is set to the new value of 06.

7. The floating point variable D is punched into the card as a 2-digit mantissa with an exponent. A decimal point is not punched. The FORMAT for D, E5.1, tells the processor that the value is in floating point mode with an exponent, that the field in the card contains 5 columns, and that there is one digit to the right of the decimal point. The object program will consider D to have the value of 12000. When the value is stored, the decimal is adjusted to the left of the high-order digit and the exponent is set to the new value of 05.

There are no more variables in the list, so columns 68 through 80 are ignored.

Specifications using the E-type format provide a great deal of flexibility. For example, consider the various methods that can be used to enter the value 10,000 into a program (1×10^4):

Punched in a card as	FORMAT specifications	Placed in storage as	
		M	E
1.E4	E4.0	10000000	05
.1E5	E4.0	Same	
1E5	E3.1	Same	
1E4	E3.0	Same	
1E6	E3.2	Same	

Output Specifications

The same FORMAT specifications of I*w*, F*w.d*, and E*w.d* are used for output statements: except that *w* now specifies the number of positions to be "reserved" for printing the number, regardless of how large the number actually is, and *d* is the number of digits to be retained to the right of the decimal point, regardless of how many digits are to the right of the decimal in the actual number in storage.

The following description of FORMAT deals with the printed line, however the principles stated also apply to paper tape and card records.

I Conversion. The specification, I4, could be used to print a number that exists in storage as a fixed point value. Three print positions would be reserved for the number and one for the sign. It is printed in this 4-space field right-justified, that is, the units position is at the extreme right. If the number in storage is greater than 3 spaces, the excess high-order positions are lost, no rounding occurs. If the number has less than three digits, the leftmost spaces are filled with blanks. If the quantity is negative, the space preceding the leftmost digit will contain a minus sign. If the quantity is plus, a blank will precede the leftmost digit.

The following examples show how each of the quantities on the left is printed according to the specification I4.

Value	Printed as
7	bbb7*
0	bbb0
−29	b−29
−3	bb−3
−146	−146
2146	b146

*b is used to indicate blank spaces

The last item is incorrect because the specification did not provide enough spaces.

F Conversion. The F-type specification is used to print a number as a floating point number **without** an exponent.

The *d* part of the format specifies the number of digits to be retained to the right of the decimal. If the number in storage has more decimal places to the right than there are places reserved for them by *d*, the extra places are truncated. If more spaces to the right of the decimal are reserved than there are decimal places in the number, zeros are filled in from the left. The numbers to the left

of the decimal are handled in the same manner as numbers converted by I-type conversion, one space is always reserved as a sign position.

Included in the count, *w*, must be a space for the decimal point and a space for the sign.

The following examples show how each of the quantities on the left is printed according to the specification F7.3.

Value	Printed as
28.601	b28.601
−6.4	b−6.400
−.8	bb−.800
4.721	bb4.721
2.48721	bb2.487

The last item is inaccurate because the specification did not provide enough spaces.

The F-type format is a convenient way of expressing the results of your computations, but it has one small pitfall. You must have some knowledge of the magnitude of the numbers you are working with. The magnitude of the number must not be so great that the size of number (the mantissa and as many decimal places as specified by the exponent) is not larger than the number of places reserved for it by your specification statement.

For example, consider the floating point number in storage

M	E
12345678	14

with the FORMAT of F10.3 (which was assumed to be large enough for this value and any other value in the series). The size of this number would be

12345678000000.

which, of course, is greater than the 10 places reserved for it. If this type of error is made, the FORTRAN program disregards the format that is specified, and instead prints the number as

b.12345678Eb14

and a message is typed on the typewriter which indicates that a floating point number is not in the allowable range of values.

The same value could have been obtained if the specification had been written E14.8 (floating point with exponent form). Of course you will get the right answer in this case, but the point is, that if you are not certain of the magnitude of your numbers, program your problem so that your answers will be printed (or punched) in floating point mode with an exponent (E conversion).

E Conversion. For E-type conversion, the *d* part of the format again specifies the number of digits to be retained to the right of the decimal. Included in the count, *w*, must be spaces for the sign and decimal point, plus four spaces for the exponent.

In 1620 FORTRAN, the object program will try to place as many significant digits to the left of the decimal that is possible by using the specification provided. Depending upon the size of the mantissa, zeros may be added to the right of the number. The position of the decimal point may be moved, and if it is, the

program automatically adjusts the value of the exponent to indicate the actual position of the decimal. The number of significant digits that will be printed can be determined by the following rules:

if $w-d \geq 8$, then 8 significant digits are printed
if $w-d < 8$, then $w-6$ significant digits are printed

The following examples show how each of the quantities on the left is printed according to the specification E10.3:

Value	In Storage	Printed
—.008	$\bar{8}000000\bar{0}\bar{0}\bar{2}$	b—.800E—02
.472	$\bar{4}720000\bar{0}\bar{0}$	b4.720E—01
.00000000006	$\bar{6}00000\bar{0}\bar{1}\bar{0}$	bb.600E—10
—10.0468	$\bar{1}00468\bar{0}\bar{0}\bar{2}$	—1.004E+01
1234567.8	$\bar{1}234567\bar{8}\bar{7}$	b1.234E+06

If your specification is not large enough, the program will automatically use the specification E14.8.

In the examples just given, it can be seen that you must know the data in order to specify a satisfactory format. Your specifications should provide for the largest number of significant mantissa digits transmitted with the greatest accuracy required.

Specifying Alphameric Fields

Alphameric data can be read into the FORTRAN program from cards, paper tape, or the typewriter. This data can be contained in the program and printed or punched as part of the output data. Alphameric fields are often used to identify totals or certain phases of the program. The following are typical output messages:

```
PROGRAM ERROR
. . . . . OHMS, . . . . . VOLTS
END OF PROGRAM
R C JONES
```

Alphameric fields require the FORMAT specification of wH, where w is the number of alphameric characters, including blanks, in the message.

The first message shown above could be printed by the following statements:

```
    PRINT 9
9   FORMAT (14H PROGRAM ERROR)
```

(The count of 14 includes a blank position before and after "program.")

The next message in the example illustrates how totals can be identified in the program. The print statement would be

```
PRINT 6, O, V
```

and the FORMAT statement might be,

```
6 FORMAT (F6.2, 5H OHMS, F6.2, 6H VOLTS)
```

The two preceding examples show how alphameric data is entered by a statement in the source program. Alphameric data can also be read from individual

cards or tape records. For example, suppose that a series of calculations is to be performed upon each customer record card in a file. To identify the results of each computation with the appropriate customer name, the following READ statement would be used:

```
READ 6, A, B, C
```

The data fields in the input card are punched A, B, customer name, and C, in sequence. The FORMAT statement therefore would be:

```
6 FORMAT (F8.2, F8.2, 14HbCUSTOMERbNAME, F8.2)
```

When the first customer card is read into the object program, the customer's name (assume it is Anderson) replaces the words "customer name" in storage. The computations for the first customer card would be printed with the PRINT statement

```
PRINT 6, A, B, C
```

and the printed line would be

```
124.16  19.14 ANDERSON      2461.25
```

Information handled with a *w*H specification is not given a variable name and cannot be referred to or manipulated in storage in any way. The maximum number of alphameric characters that can be specified is 49.

Blank Field Specification

Skipping a blank field in input data was shown in an earlier example. Blank characters may be provided in an output record with the same specification, *w*X. The FORMAT statement

```
6 FORMAT (10X F10.3, E14.8)
```

would provide 10 blank spaces before the first value is printed. A comma is not required after the blank field specification.

The maximum blank field specification is 49, but two specifications may be written in succession to provide more than 49 blank positions.

Multiple Use of Single Specifications

It was stated earlier, that each variable listed in an input or output statement must have a corresponding specification provided in a FORMAT statement. However, one specification could be used for one or more variables in an input list, **if all items in the list required the same specification.**

For example, a READ statement containing six variables, all requiring the same format specification, could use the FORMAT statement

```
1 FORMAT (E8.2)
```

The object program processes all input and output (I/O) statements by (1) scanning the I/O statement to get the name of the variable, and (2) scanning the FORMAT statement to get the specification for the variable. It repeats this process until all variables have been processed. When the program has reached the last specification in the FORMAT statement, and there are variables in the I/O statement that have not yet been processed, the program returns to the last open parenthesis in the FORMAT statement and continues to scan the next specification in sequence from left to right. The program will use the FORMAT specifications

repeatedly (always returning to the last open parenthesis) until all variables in the input or output statement have been processed. Each time the program returns to the last open parenthesis, the input or output record is terminated. In output operations, this means that a new card or paper tape record is punched containing the remaining items on the list. In input operations, a card or paper tape record cannot contain more items than there are specifications in the FORMAT statement. Thus, the input or output statement is completed when there are no items remaining on the list.

If there is a long list of data to be printed, the statement

```
8 FORMAT (F10.6,E10.2,(E8.4,I3))
```

is the same as writing the statement

```
8 FORMAT (F10.6,E10.2,E8.4,I3/E8.4,I3/ . . .)
```

In this example, the first printed line would contain the first four variables in the PRINT list, with the format of F10.6, E10.2, E8.4, and I3. All remaining variables in the PRINT list would be printed on succeeding lines, two to a line, in the format E8.4, I3. As explained next, the use of a slash symbol (/) as a special character makes it possible to print on more than one line.

Printing Multiple Lines with One Format Statement

A list of variables in a PRINT statement can be printed on one or more lines by placing a slash between the specifications. For example, a list of four variables with the FORMAT statement of

```
6 FORMAT (F10.2,F10.2/E10.4,E10.4)
```

would be printed with the first two variables on the first line, and the last two variables on the next line.

A great deal of flexibility can be obtained in specifying multiple-line printing. Consider the following statements:

```
PRINT 3, A,B, . . . ,Z
3 FORMAT (F9.2,F10.4/E14.5)
```

(In an actual program, each item from A to Z would have to be listed.) When the output data is printed, lines 1, 3, 5 . . . have format (F9.2,F10.4), and lines 2, 4, 6 . . . have format (E14.5).

Notice that both the slash and the closing parenthesis in a FORMAT statement indicate the termination of a "record." This is not too significant when you are printing on the typewriter because a "record" is merely a typewritten line. If you are using card output, the end of a record means the end of punching in one card and the remaining variables are punched in the next card. If you are using paper tape output, the termination of a record means that an end-of-line character is punched into the tape and the remaining variables are punched into the following tape record.

Blank lines can be included in typewritten output by inserting slashes into a multiline FORMAT. N + 1 consecutive slashes produce N blank lines if it is included between two specifications. N slashes before the first specification, or after the last specification produces N blank lines (using the slash in card and paper tape output is possible, but of limited value).

Some Review and Additional Thoughts About Format and Input/Output Statements

1. Specifications in a FORMAT statement **must** be in the same mode (fixed point or floating point) as the corresponding items in the input or output list. For example:

```
PRINT 2,A,B,J
2 FORMAT (F8.2,F8.3,I8)
```

2. If a FORMAT statement specifies more characters to be printed or punched than there are positions in the output record, the excess characters are lost.
   a. A typewritten line has a maximum of 87 characters.
   b. A punched card has a maximum of 72 positions.
   c. A paper tape record has a maximum of 87 characters.
3. In input statements, I-type data must be located at the extreme right (to avoid truncating pertinent data).
4. In an input statement, minus or plus signs must occupy a separate column of the record. Plus signs may be indicated by a plus symbol or a blank. A number without a sign position is assumed to be plus. Blanks in numerical fields are regarded as zeros.
5. Numbers for E-type conversion need not have four columns devoted to the exponent field. The start of an exponent field must be marked by an E, or if that is omitted, by a + or −, but not a blank. Thus, E2, E02, +2, +02, E02, and E+02 are all permissible exponent fields, and must always be right-justified. Whichever of these forms you use, it is suggested that you be consistent in using the same one.

DIMENSION Statement

Whenever you use subscripted variables in your program, you must provide the processor with the following information:

1. Which variables (of all the variables you may have used in your program) are **subscripted.**
2. Whether your subscripted variables (arrays) are one- or two-dimensional.
3. The number of elements in each array.

The DIMENSION statement provides information to the processor necessary for the allocation of storage in the object program for the elements of arrays of quantities. One DIMENSION statement may be used to dimension any number of arrays, as long as the entire DIMENSION statement does not exceed the length of a statement (72 characters).

Format	"DIMENSION *v(d), v(d,d), v(d)*"...for one- and two- dimensional arrays. where each *v* is the name of a variable, followed by parentheses enclosing one or two constants, representing the number of elements in each array. The *v*s must be separated from each other by commas. The constants must be unsigned and in fixed point mode.
Examples:	DIMENSION HRS (12) DIMENSION A(10), B(10, 5)

Every variable which appears in the program in subscripted form must appear in a DIMENSION statement, and the DIMENSION statement must precede the first appearance of the variable. When the object program is processed, the number of elements in an array must not be larger than the number specified in the DIMENSION statement. In the first example shown, the variable, HRS, is an array

consisting of 12 elements, and the processor will set aside twelve 10-position fields of storage (this is, a floating point variable – 8 for mantissa and 2 for characteristic). In the second example, the variable, B, represents a two-dimensional array that will consist of 10 rows with 5 columns in each row. The processor will set aside fifty (10 x 5) 10-position fields in which to store the elements of the array B.

You may include both fixed point and floating point variables in the same DIMENSION statement. The DIMENSION statement does not create instructions in your object program, its function is merely to supply information to the processor.

A FORTRAN Problem

The problem contained in this section is intended as a guide for developing your first FORTRAN problem. Rather than try to show the power of FORTRAN, a simple, uncomplicated problem was chosen. It indicates how a problem is developed, how it is written on the coding form, and how it is documented as it is processed at compile time and object time.

Block Diagramming

Diagramming is a technique of schematically showing the steps which the computer must take to produce the answers required by the problem.

Diagrams serve two purposes:

1. They offer an easy notation for analyzing the steps required in the solution of a problem.
2. They provide the basic documentation in the form of a "map" of the program, so that someone unfamiliar with the program can easily determine what the program does and how it does it.

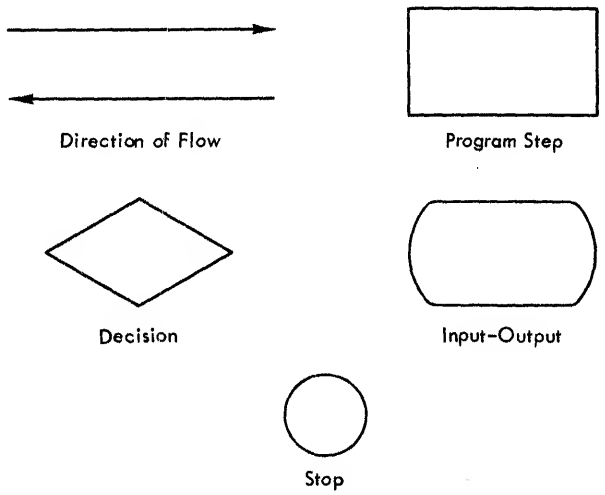
It is for these reasons that diagramming is not only highly recommended, but is often required at data processing installations.

Techniques of diagramming vary greatly, as do the symbols used. In addition, diagramming may be very general, or extremely detailed to the point where every machine instruction is included.

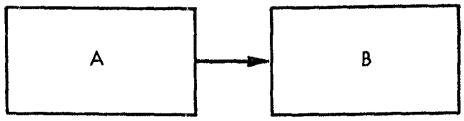
The more complete the diagram, the easier the job of actually writing the program; however, initial analysis of a problem can usually be noted only in major steps.

Only simple diagramming techniques will be explained here. Further details of the technique are available in the IBM *Reference Manual, Flow Charting and Block Diagramming Techniques* (Form C20-8008).

The symbols to be used are explained below:

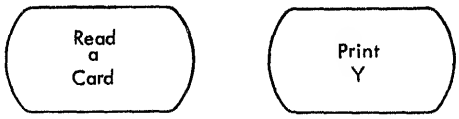


The Direction of Flow symbol simply shows the relationship between symbols.

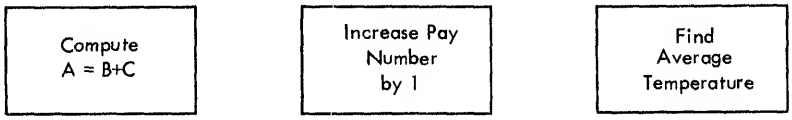


The example shows that A is executed first, then B.

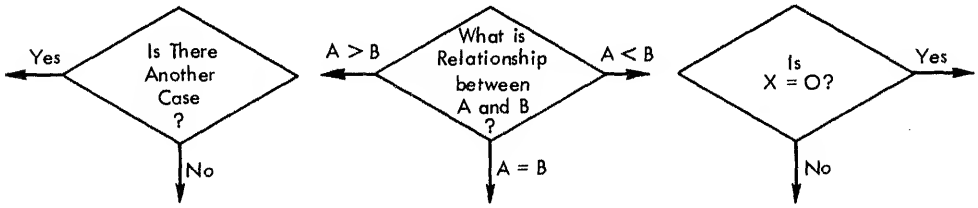
The Input/Output symbol is used to refer to any operation that involves an input/output device.



The Program symbol is used to represent any steps in the program which are not represented by special symbols.



The Decision symbol represents any logical decision that is contained in the program.



The Stop symbol is used to indicate the end of the program.

Diagramming the Problem

Problem: Solve for f(x)

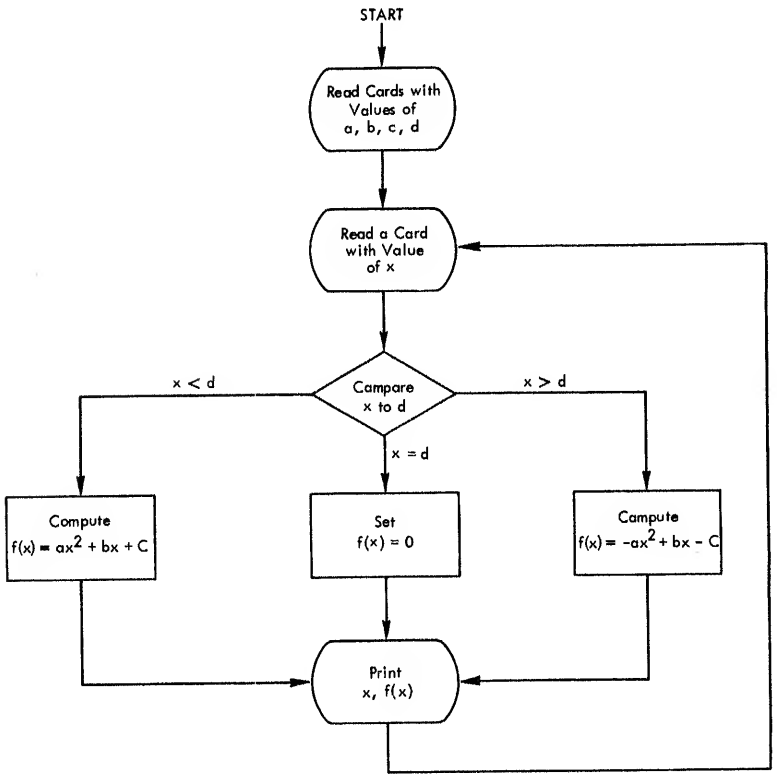
Given: Values for a, b, c, and d punched on a card, and a set of values for the variable x punched one per card

Evaluate the function defined by

$$f(X) = \begin{cases} aX^2 + bX + C & \text{if } X < d \\ 0 & \text{if } X = d \\ -ax^2 + bX - C & \text{if } X > d \end{cases}$$

for each value of x, and print the value of x and f(x).

A block diagram of a possible FORTRAN program to solve this problem follows:



Writing the FORTRAN Statements

The FORTRAN statements to solve this problem are shown in the coding chart which follows. In this problem, statement numbers required by the logic of the program are either 1 or 2 digits; statements with 3-digit numbers are numbered only for the purpose of explanation here, and would not need to be numbered in an actual program.

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–55)
C		FUNCTION OF X PROBLEM
100		READ 7,A,B,C,D
6		READ 7,X
101		IF (X-D) 2,3,4
2		FOFX = A*X**2.+B*X+C
102		GO TO 5
3		FOFX = 0.
103		GO TO 5
4		FOFX = -A*X**2.+B*X-C
5		PRINT 1,X,FOFX
104		GO TO 6
1		FORMAT (F14.5,F14.5)
7		FORMAT (F4.0)
		END

The first statement is a comment which will appear on source program listings. A comment statement must be identified by placing a C in column one of the coding form.

Statement 100 causes the first card to be read and the values punched in that card to be assigned as the values of A, B, C, and D. This statement references FORMAT statement 7, which specifies that each field of the card is four columns long, and that each value has a decimal point punched into the card.

Statement 6 causes the next card to be read, and references FORMAT statement 7. Statement 6 contains the first value of x to be used by the program.

Statement 101 determines the relationship between X and D and determines which formula to use in the computation of f(x). If X—D is negative (X < D), the program is transferred from 101 to statement 2; if X — D is zero (X = D), the program is transferred from 101 to statement 3; if X — D is plus (X > D), the program is transferred from 101 to statement 4.

Statements 2, 3, or 4 are used to determine the correct value of f(x); i.e., FOFX. Regardless of which of the three computations occurs, the program is always transferred to statement 5.

Statement 5 types out the values of x and f(x) and references FORMAT statement 1. This FORMAT specifies two 14-position fields; each field contains five positions to the right of the decimal point.

Statement 104 causes the program to transfer to statement 6 to read the next value of X, and the pattern continues until all of the X cards have been processed.

The computer will automatically stop when it attempts to execute the READ statement and there are no more cards in the card reader.

Statement 1 and 7 contain the FORMAT specifications for the input and output statements.

The END statement indicates to the processor that the source program is completed.

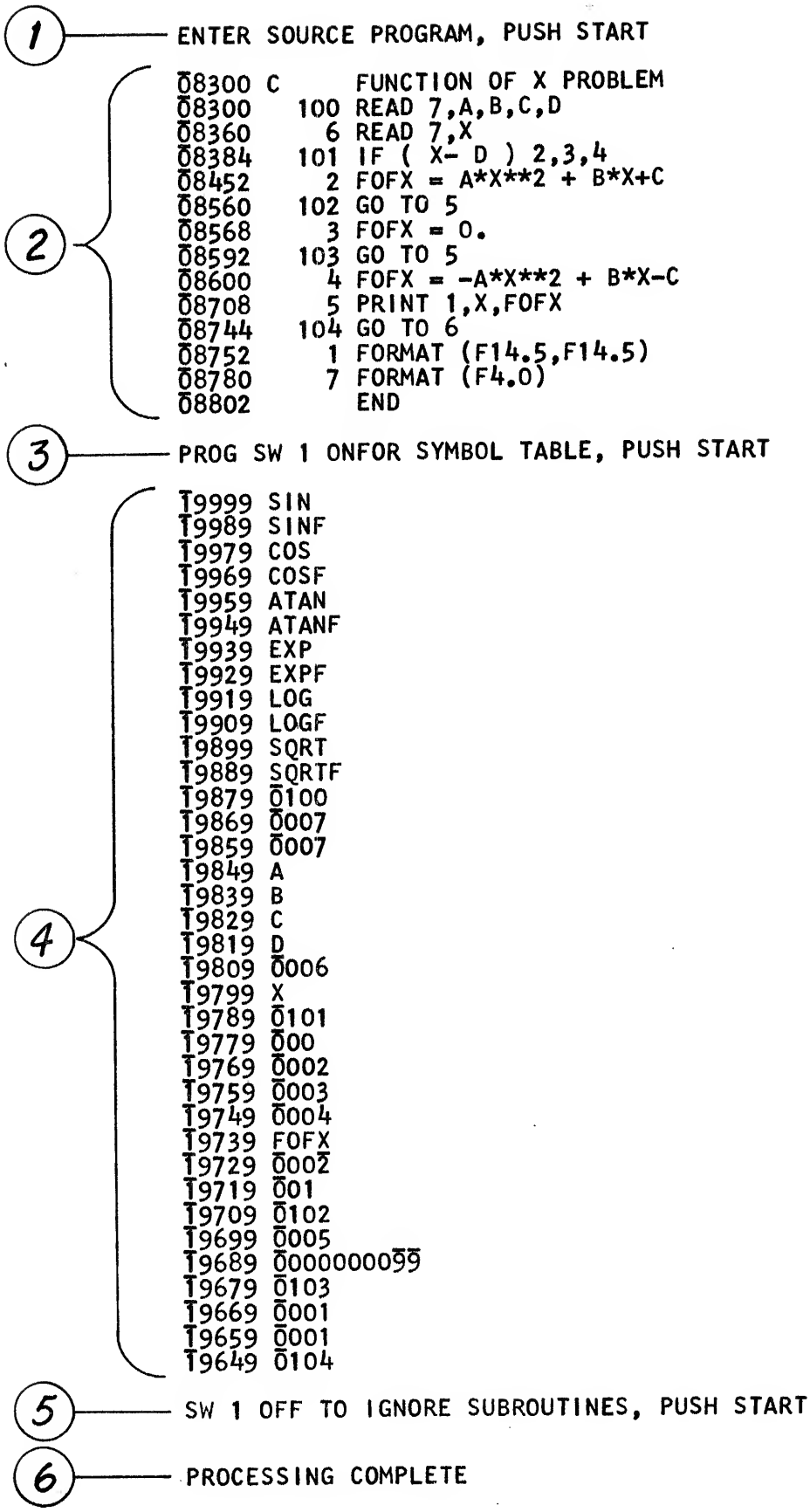
Processing the Source Statements

The operating procedure for processing a source program is given in OPERATING PRINCIPLES. The information contained here concerns only the documentation that is available as the object program is compiled or processed.

The following illustration shows the typed output that can be prepared for the function of x problem.

Numbers have been drawn in the left margin of the typeout to indicate the phases of processing for this explanation.

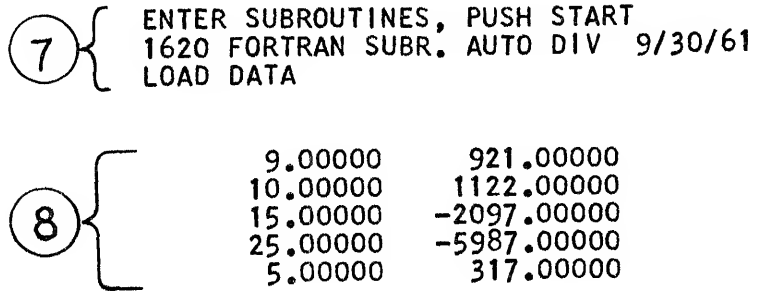
1. After the processor is loaded into the 1620, this message is typed.
2. As each source statement is compiled it can be typed. The 5-digit address is the starting address in core storage for instructions compiled for that statement.
3. After the source statements have been compiled, the programmer or operator can have the symbol table typed.
4. A symbol table can contain the storage addresses for the subroutines used; it can also contain storage area for constants and variables, variable arrays, temporary accumulators, and statement numbers. (This subject is covered in more detail in Part 4, ANALYSIS OF THE FORTRAN PROGRAM.)


1
ENTER SOURCE PROGRAM, PUSH START
2
08300 C FUNCTION OF X PROBLEM
08300 100 READ 7,A,B,C,D
08360 6 READ 7,X
08384 101 IF (X- D) 2,3,4
08452 2 FOFX = A*X**2 + B*X+C
08560 102 GO TO 5
08568 3 FOFX = 0.
08592 103 GO TO 5
08600 4 FOFX = -A*X**2 + B*X-C
08708 5 PRINT 1,X,FOFX
08744 104 GO TO 6
08752 1 FORMAT (F14.5,F14.5)
08780 7 FORMAT (F4.0)
08802 END
3
PROG SW 1 ONFOR SYMBOL TABLE, PUSH START
4
19999 SIN
19989 SINF
19979 COS
19969 COSF
19959 ATAN
19949 ATANF
19939 EXP
19929 EXPF
19919 LOG
19909 LOGF
19899 SQRT
19889 SQRTF
19879 0100
19869 0007
19859 0007
19849 A
19839 B
19829 C
19819 D
19809 0006
19799 X
19789 0101
19779 000
19769 0002
19759 0003
19749 0004
19739 FOFX
19729 0002
19719 001
19709 0102
19699 0005
19689 0000000099
19679 0103
19669 0001
19659 0001
19649 0104
5
SW 1 OFF TO IGNORE SUBROUTINES, PUSH START
6
PROCESSING COMPLETE

Processing the Object Program

5. In this program, the object program was processed immediately following its compilation, therefore the subroutines were not assembled as part of the object program, but instead, were loaded into the 1620 after the object program was loaded.
6. "Processing complete" means the object program has been executed.

The illustration shows the typeout information that results when the object program is processed with data for various values of x, and A, B, C, and D have the values of 10.0, 11.0, 12.0, and 13.0 respectively.



7. The subroutines are placed into the 1620, read into storage, and the message "Load Data" is typed.
8. The data is loaded. Five values of x are shown with their corresponding values of f(x).

Program Testing

After you have written your program, you should thoroughly "desk check" it. Desk checking is the process of looking over the source program for obvious mistakes in logic or form, and the actual **manual** run of an item of data through the program. This technique will quite often turn up a surprising number of errors in a new program.

After you have desk checked your program, you should prepare test data that can be processed on your program. The test data should be accompanied by the correct results so that they can be compared by the machine operator to the results obtained when your program is tested.

You should devote enough time for a careful selection of test data which will check out the various and numerous combinations that may exist in the logical flow of your program. It is advisable to start out with items of data which will produce the simplest logical flow through your program and then to follow with items of data which will take increasingly more complex paths, so that each new item will involve the use of a new subroutine or segment. The more careful your data selection is, the less chance there will be of errors in actual data results.

Program Verification

When your object program is processed with the test data, and the results indicate that there is an error in the program, you can do several things to locate the difficulty.

1. Check the test data to be certain it is punched (or typed) exactly as you think it is punched. This is also true for actual input data. Keep in mind that persons preparing data for your program may occasionally make common errors peculiar to the format you are using.
2. If possible, have an associate make a desk check of your program, or if the program was typed out when it was compiled, check that program.

3. During the compilation of an object program, a number of tests are made for source program errors. If an error is found in a source statement, an error message is typed and the processing continues. You should determine if an error message was typed out and not noticed by the machine operator. These error tests are concerned with the violation of the rules for forming expressions and statements. (The meaning of the error message typed out is described in Part 3, OPERATING PRINCIPLES.)
4. You can make a more thorough test of your source program by processing it on the IBM FORTRAN Pre-Compiler program which is available from IBM upon request. The Pre-Compiler program detects and permits correction of errors in a FORTRAN source program **before** it is compiled. It detects many of the more common programming errors and indicates possible logical errors in the source program as a whole. If you do not have time to use the Pre-Compiler program, a knowledge of the types of errors which can be detected by this system may help you to visually locate errors in the source program. The Pre-Compiler program is described in Part 5. The Pre-Compiler description contains a list of 51 of the most common programming errors.
5. If you still haven't located the difficulty, you can use the "trace feature" of the FORTRAN program. The FORTRAN processor can compile certain instruction into an object program which will permit tracing the flow of the object program in order to check its correctness. When the object program is executed, the trace output consists of the evaluated left-hand side of each executed arithmetic statement.

   To use the trace feature, you would have to compile the program again with the trace feature instructions included and execute the object program using this feature. Checking your program with the trace feature is time consuming, but it will locate the errors in your program. The following illustration shows how the results of the function of x problem would look if the trace feature had been used.

```
ENTER SUBROUTINES, PUSH START
1620 FORTRAN SUBR. AUTO DIV  9/30/61
LOAD DATA


 .92100000E+03
       9.00000      921.00000
 .11220000E+04
      10.00000     1122.00000
-.20970000E+04
      15.00000    -2097.00000
-.59870000E+04
      25.00000    -5987.00000
 .31700000E+03
       5.00000      317.00000
```

Additional information about the trace feature is provided in Part 3, OPERATING PRINCIPLES.

Part 3—Operating Principles

This part of the manual provides the information necessary for implementing the FORTRAN program on the 1620 computer. It is assumed that the reader has a prior knowledge of 1620 operating principles. OPERATING PRINCIPLES is divided into two parts, PRODUCING THE OBJECT PROGRAM and EXECUTION OF THE OBJECT PROGRAM.

Producing the Object Program

The FORTRAN program is available in two forms, card or paper tape. Both forms are divided into two sections; the processor and the subroutines. The sequence of operations that follows is written for both card and paper tape systems.

Eight basic steps are required for producing the object program. These eight steps are summarized below, followed by additional detailed information for steps 1, 2, 6, 7, and 8.

1. Clear core storage to zeros.
2. Set the console program switches for compilation.
3. Set the overflow check switch to PROGRAM and all other check switches to STOP.
4. Press the reset key.
5. For the card system, prepare the card punch for operation by loading blank cards into the punch hopper and by pressing the punch start key. For the paper tape system, prepare the paper tape for operation.
6. Load the compiler program deck or tape.
7. Enter the source program statements. These may be read in through the card reader, the paper tape reader, or typed in at the console typewriter.
8. If required, load the subroutine deck or tape.

(Step 1)

Clearing Core Storage to Zero

A suggested method for clearing core storage to zeros is:

1. Press the reset key.
2. Press the insert key.
3. Type the instruction 16 00010 00000.
4. Press the release key.
5. Press the start key.
6. After all storage positions have been cleared, press the instant stop key.

(Step 2)

Switch Settings

During compilation of the source program, the console program switches perform the following functions:

	ON	OFF
Switch 1	Causes the source statements to be typed on the console typewriter as they are processed. The first 5-digit field is the object program address of the first instruction compiled for the source statement.	Source statements are not listed.
Switch 2	Causes trace instructions to be compiled.	Trace instructions are not compiled.

Switch 3	Input to the compiler (source statements) is being entered via the console typewriter.	Source program entered from card reader or paper tape reader.

Switch 4 This switch is used in conjunction with switch 3 when switch 3 is ON. It provides the ability to restart the typing of a statement if you have made an error. Switch 4 is normally OFF. When a typing error is made in a source statement and it is to be corrected, this switch is turned ON, the release and start keys are pressed, and then switch 4 turned OFF. The statement can now be retyped.

Loading the Compiler

(Step 6)

Card System

When operating with the card system, you can load the compiler program deck by placing the deck in the read hopper and pressing the load key. The cards in the FORTRAN compiler deck are punched with sequence numbers in columns 76 through 80 and must be loaded in sequence. If the first card read is not card number 1, the machine will stop with an operation code of 00 displayed in the operation register lights. If cards 2 through 24 are not read in the proper sequence, the message "CARDNN," where NN is the number of the missing card, will be typed on the console typewriter and the machine will halt. The cards must be removed from the reader and placed in proper order. Core storage must be cleared to zeros before the deck is read in again starting with card 1.

Beginning with card number 25, if any card is out of sequence, the console typewriter carriage will be returned and the following message will be typed:

CARD $\bar{0}$NNNN OUT OF SEQUENCE

and the machine will halt. When this occurs, the card numbered $\bar{0}$NNNN has been read out in sequence. Remove the cards from the reader and arrange them correctly. Starting with the card replacing card number $\bar{0}$NNNN, put that part of the deck which has not yet been loaded, back into the read hopper. Press the reader start key on the 1622, and continue reading by pressing the start key on the 1620 console.

Paper Tape System

To load the compiler tape, the following procedure must be followed:

1. Mount the compiler tape on the paper tape reader.
2. Press the insert key.
3. Type the instruction 36 00000 00300.
4. Press the release key.
5. Press the start key.

The following instructions are typed on the console typewriter after the compiler has been successfully loaded:

ENTER SOURCE PROGRAM, PUSH START

Compilation of the Source Program

To begin compilation after the compiler has been loaded, either press the start key or manually insert the instruction 49 00402.

Two methods of source program input may be used under control of program switch 3, as follows:

1. If input is in cards (switch 3 off), place the source program deck in the read hopper and press the reader start key. If input is in paper tape, mount the source program tape on the paper tape reader.

2. If the source program is to be entered from the typewriter (switch 3 on), the compiler will transfer control to the console to await the first statement. After you type a statement, press the record mark key and then press the release and start keys to continue compilation. The carriage will return after each statement has been processed, to await the entry of the next statement until an END statement is entered.

As the source program is processed, a test is made in the compiler to determine whether the compiled object program (not including relocatable subroutines), together with the object program data table, will occupy more core storage locations than will be available. If the object program is too long, the statement which caused the overlap is processed, and the following message is typed immediately:

OVERLAP

Compilation continues, and the message is typed after each statement thereafter.

After an END statement is processed, the following instruction message is typed on the console typewriter:

SW1 ON FOR SYMBOL TABLE, PUSH START

If a typed listing of the symbol table, that was developed during compilation, is not desired, turn off program switch 1. If the listing is required, turn on switch 1.

To continue processing, press the start key.

The following message is typed next, whether the symbol table has been typed or not.

SW1 OFF TO IGNORE SUBROUTINES, PUSH START

If the subroutines are to be included in the object program deck or tape, turn on program switch 1, load the subroutine deck or tape, and press the start key. If the subroutine deck or tape is to be read in when the object program is run, turn off switch 1.

To complete the processing, press the start key.

If program switch 1 is off, the following message will be typed:

PROCESSING COMPLETE

(Step 8)

Loading the Subroutines

Under control of program switch 1, as previously described, the subroutine deck or tape may be loaded immediately after compilation, or loaded when the object program is loaded.

Card System

When operating with the card system, place the subroutine deck in the read hopper and load the deck into storage by pressing the start key. (Or you may press the reset key on the console, and then press the load key on the card reader.)

The cards in the subroutine deck have a sequence number punched in columns 76 through 80, and must be loaded sequentially in that order. If cards numbered 1 through 8 are not read in proper sequence, the machine will halt with an invalid operation code displayed in the operation register lights. If this occurs, remove the cards from the reader, place them in the proper sequence, and replace the deck in the read hopper. Press the reset key and then the load key.

Cards out of sequence, other than cards 1 through 8, will cause the message

CARD OUT OF SEQUENCE

to be typed on the console typewriter, and the machine will halt. The second card from the back in the read stacker is the one out of sequence. All preceding cards were loaded properly. Remove the cards from the reader which have not been loaded, arrange them sequentially, and replace the deck in the read hopper. To continue reading the subroutine deck, press the reader start and start keys.

Paper Tape System

When operating with the paper tape system, mount the subroutine tape, and load it by pressing the start key. (When restarting, you may insert the instruction 36 00000 00300, press the release key, and then press the start key.)

If the source program has used any of the relocatable subroutines, they will either be punched out into the object program if the subroutines are read in immediately after compilation, or they will be loaded into core storage if the subroutines are processed at object time.

If the compiled instructions and required data will exceed the storage capacity of the 1620 at object time, the following message will be typed on the console typewriter just after the first relocatable subroutine that causes the overlap has been processed for possible inclusion in the object program:

OVERLAP XXXXX POSITIONS

The "OVERLAP XXXXX" is the number of core storage positions which overlap between the end of the object program and the data storage area. The object program is allocated to increasing core storage locations and includes the compiled program and relocatable subroutines used. The data storage area is allocated to decreasing core locations starting with the highest addressed position of storage. After the overlap message is typed, the machine will halt and programmed processing of the subroutines cannot be continued.

After the subroutines have been processed, the following message will be typed on the console typewriter:

PROCESSING COMPLETE

Errors in the Source Program

A number of tests are made for source program errors during compilation of the object program. If an error is found in a source statement, an error message is typed, "ERROR NO. *n*," where *n* is the error code; and processing continues. A list of possible errors follows:

Error No.	Condition
1	An incorrectly formed statement.
2	A subscripted variable is used and no DIMENSION statement for it has previously appeared in the program, or a dimensioned variable is used without subscripts, or a variable used in a DIMENSION statement has already appeared in the source program.
3	A floating point number is not in the allowable range of values, or a fixed point number contains more than four digits.
4	The symbol table is full.
5	A mixed mode expression (fixed point and floating point in the same expression.)
6	A variable name in an expression containing more than five characters.

7 The switch number has been omitted in an IF (SENSE SWITCH n) statement, or the first character following the right parenthesis in an IF statement is a comma.
8 A comma follows the statement number in a DO statement.
9 A DIMENSION statement ends with a comma, or more than two dimensions have been specified in a DIMENSION statement. (Only two-dimensional arrays are permitted.)
10 Unnumbered FORMAT statement.
11 Incorrect representation in a FORMAT statement in one of the following ways:
   a. A special character is used
      = @ — * $ + . ,
      in a numerical field specification.
   b. An alphabetic character other than E, F, or I is used in a numerical field specification.
   c. A decimal point is missing in an E- or F-type numerical field specification.
   d. The number of positions to the right of the decimal point has not been given in an E- or F-type numerical field specification.
   e. A record mark appears in a numerical field specification or in an alphameric field.
   f. The first character following the word FORMAT is not a left parenthesis.
12 The total record width specified in a FORMAT statement is greater than 87 characters.
13 A FORMAT statement number has been omitted in an input/output statement.

Compilation of the program proceeds after the error message is typed, but the statement in which the error has occurred may either be partially compiled or not compiled at all.

Execution of the Object Program

Card System

When operating with the card system, the object program may be loaded immediately after compilation by placing the deck in the read hopper, pressing the reader start key and the start key on the console. The object deck may also be loaded at this or any other time by first pressing the reset key and then the load key on the card reader.

The cards in the object program must be loaded, sequentially by number, starting with number 0001 which is punched in columns 77 through 80. If cards numbered from 1 through 8 are not read in proper order, the machine will halt with an invalid operation code (00) displayed in the operation register lights. The cards must then be removed from the reader and placed in proper sequence. Reload by placing the cards in the read hopper again and by pressing the reset and load keys.

Any other card out of sequence will cause the message

CARD OUT OF SEQUENCE

to be typed on the console typewriter, and the machine will halt. The procedure for continuing the operation is exactly as described for reloading the subroutine deck under similar conditions.

Paper Tape System

When operating with the paper tape system, the object program may be processed immediately after compilation by mounting the object tape and pressing the start key.

The object tape may also be entered by pressing the insert key, typing the instruction 36 00000 00300, and pressing the release and start keys.

If the subroutine tape or deck is to be loaded at object time, after the object program has been loaded, the machine will halt and the following message will be typed on the console typewriter.

ENTER SUBROUTINES, PUSH START

The subroutine deck or tape must then be loaded in the manner already described.

After the subroutines have been loaded, the machine will halt and the following message will be typed:

LOAD DATA

If the subroutines are already contained in the object deck or tape, the following message will be typed after the object program has been loaded, and the machine will halt:

LOAD DATA

To initiate the execution of the object program, press the start key on the 1620 console, or manually insert the instruction 49 08300.

Input Data From the Keyboard

Each execution of an ACCEPT statement in the object program causes the typewriter carriage to return as a signal for you to type the input quantities corresponding to the variables named in the list. If you make a typing error during console entry of data, you may correct the error by using program switch 4, as described under SWITCH SETTINGS.

NOTE: When typing data from the keyboard, the total width specified in the FORMAT specification should be typed. Leading or trailing blanks may be used to fill out a specification.

Trace Feature

The FORTRAN processor can (under program switch control) compile certain instructions into the object program for tracing the flow of the program and for checking its correctness. When the object program is executed, program switch 4 performs the following function:

	ON	OFF
Switch 4	Causes compiled trace instructions to be executed.	Trace instructions are not executed.

The trace output provided is the evaluated left-hand side of each executed arithmetic statement, which is typed at the left margin. Normal output, resulting from PUNCH, PUNCH TAPE, PRINT, and TYPE statements is not inhibited. The output format of the trace data is E14.8 for floating point results and I5 for fixed point results.

Note that program switch 4 serves a dual function during execution of the object program: i.e., provision of trace data and correction of input data incorrectly entered at the console keyboard. Thus, when running in the trace mode, you must turn off program switch 4 before typing output data. Following the entry of the last item on the input list (after pressing the record mark key and release key), press SIE two or three times, turn the switch on, and press the start key. (If a trace routine is desired, switch 4 cannot otherwise be used in the program.)

Part 4—Analysis of the FORTRAN Program

This part of the manual is intended to assist experienced programmers in understanding, modifying, and testing FORTRAN programs. It is assumed that the reader has had previous experience in programming, and has some knowledge of the 1620 Symbolic Programming System, if subroutines other than those provided by FORTRAN are to be added.

Subroutines

The FORTRAN subroutine deck or tape contains thirty-one subroutines. Up to nineteen additional subroutines may be added at the user's option. It is entirely feasible for several subroutine decks or tapes to be maintained by an installation when it is desirable to have several sets of optional subroutines available.

Two hardware-oriented systems are available; one for use on machines having the automatic divide feature, and one for machines which do not have automatic divide.

The subroutines are classified as follows:

Type A: Automatically compiled, used by the FORTRAN system only; not directly available to the programmer.

Type B: Automatically compiled if used in the source program, or used by the system, available to the programmer.

Type C: Not used by the system, automatically compiled if used by the programmer.

The table which follows shows each subroutine provided, its type, the number of operands it requires, and its symbolic name. The symbolic names shown on this table are not used in programming; they are included to provide reference to the symbolic listing of the subroutines.

Subroutine	Symbolic Name	Operation	Type
Floating Add	FAD	A + B	A
Floating Subtract	FSB	A - B	A
Floating Multiply	FMP	A * B	A
Floating Divide	FDV	A / B	A
Reverse Floating Divide	FDVR	B / A	A
Floating A**B	FAXB	A**B	A
Floating A**B(-B)	FAXBN	A**(-B)	A
Fixed Add	FXA	I + J	A
Fixed Subtract	FXS	I - J	A
Fixed Multiply	FXM	I * J	A
Fixed Divide	FXD	I / J	A
Load Into FAC	TOFAC		A
Store from FAC	FMFAC		A
Reverse Fixed Divide	FXDR	J / I	A
A**I	FAXI	A**I	A
A**(-I)	FAXIN	A**(-I)	A
Convert Sign	RSGN	-A	A
Floating Natural Log	FLN	LOG(A)	B
Floating Exp(A)	FEXP	EXP(A)	B
Floating Square Root	FSQR	SQRT(A)	C
Floating Sine	FSIN	SIN(A)	C
Floating Cosine	FCOS	COS(A)	C
Floating Arctangent	FATN	ATAN(A)	C
Convert: float-to-fix	FIX	FIX(A)	A
Convert: fix-to-float	FLOAT	FLOAT(I)	A
Read Card	RACD		A
Read Tape	RAPT		A
Read Typewriter	RATY		A
Write Card	WACD		A
Write Tape	WAPT		A
Write Typewriter	WATY		A

Floating Point Accumulator

The results of all floating point subroutines appear in a 10-digit field which extends from storage positions 00051 through 00060. This field is called the floating point accumulator (FAC). The symbol, FAC, is associated with the address 00060 in the symbolic listing of the system.

FAC is also used as the fixed point accumulator. Fixed point numbers occupy only the four low-order positions of FAC, 00057-00060.

Subroutine Linkage

The subroutine linkage is in the form

BTM SUBR A (where A is the address of the argument)

for arithmetic subroutines. The operand is added to, subtracted from, divided by, or multiplied by the number stored in FAC.

Both type B and type C subroutines are relocatable and are loaded only if called for. Toward the beginning of the compilation phase, the symbol table area is cleared. The symbolic name of the subroutine, SIN, is loaded into a specific 10-digit field in the symbol table. The address of this field is derived from the order in which the subroutine names are listed in the FORTRAN processor. The symbol is left-justified in the field, and the high-order address of that field is associated with the function subroutine. The 10-digit field immediately preceding this field is also associated with the same function subroutine. For example, if the subroutine order is the order used by IBM Applied Programming in the decks they prepare and release, the locations 19990 through 19999 are reserved for the symbol SIN and the preceding ten digits, 19980 through 19989, are also reserved for the sine subroutine. If the program calls for the sine subroutine (i.e., the sine function is used in an arithmetic statement), the following instruction is compiled:

BTM 19990 , A

where A is the address of the subroutine argument. When the subroutine has been assigned an absolute address, the symbol SIN is replaced by 49 xxxxx, where xxxxx is the absolute address of the FSIN subroutine in memory. Thus, when the BTM 19990 A instruction is executed in the object program, the address of the argument will be transmitted to 19985 through 19989, and the branch to 19990 will be followed by a branch to the FSIN subroutine.

Error Analysis of Subroutines

Results of all FORTRAN subroutines are truncated (except FMP and FEXP, where the result is rounded), and, in general, errors are no greater than one in the last digit of the resulting mantissa. The exceptions to this statement are listed below.

FLN: The argument of the FLN subroutine is broken into an integral and a fractional part. The logarithm of the fraction is evaluated using a series expansion. The result is correct to nine decimal digits. The integral part of the argument is multiplied by ln10 and added to the above result to produce the desired value. For values of the argument in the range $.99 < ARG \leq 1.01$, some loss of accuracy will occur. The reason for this is that some of the digits calculated will be leading zeros, and, when the result is normalized, fewer than eight significant digits will remain.

FEXP: The antilogarithm is computed using a Hastings' approximation* for 10^x. The argument is initially multiplied by log e and then divided into an integral and a fractional part. The integral part becomes the characteristic of the re-

*Hastings, Cecil, Jr., *Approximations for Digital Computers*, Princeton University Press, New Jersey, The Rand Corporation, 1955

sult; the fractional part is evaluated in the polynominal to produce the mantissa. When the argument of the function is positive, the error in the result does not exceed one in the last digit of the mantissa; when the argument is negative, the limit of error is five in the last digit of the mantissa.

FAXB and FAXBN: A^B is evaluated as $e^{B \ln A}$; therefore, it is evaluated by means of three linking subroutines, FLN, FMP, and FEXP. An error in one of these subroutines may propagate and increase in succeeding subroutines. An effort is made to counteract this effect by rounding the product BlnA in the FMP subroutine. The error thus produced is in general no greater than one in the seventh digit of the mantissa.

FSQR: The square root is computed by means of the odd integer method. The result is accurate to 1 in the last digit of the mantissa.

FSIN and FCOS. The sine and cosine functions are computed using a Hastings' approximation for sine

$$\frac{\pi}{2} X$$

Before it can be used, this approximation is transformed to compute sine X for

$$-\frac{\pi}{2} \leq X \leq \frac{\pi}{2}$$

and cosine is evaluated as the sine

$$\frac{\pi}{2} - X$$

The result of this subroutine is correct to eight decimal digits. However, for arguments less than or equal to one-tenth of a radian, leading zeros in the result will cause a loss of accuracy upon normalization, as with FLN. Loss of accuracy will result for arguments larger than 4 π and less than 100 radians, but will not exceed one in the seventh digit of the mantissa. The reason for this is that the larger the number of radians, the less accurately the angle can be specified when reduced to within one revolution. For arguments greater than 100 radians, correspondingly greater errors will be produced.

FATN: The arctangent function is evaluated by using the first six terms of a series expansion, which results in an error of less than one in the last digit of the mantissa. In the computation, arctan x must be in the range

$$-\frac{\pi}{2} \leq \arctan x \leq \frac{\pi}{2}$$

If $|x| < 1 \times 10^{-4}$ the resulting angle is equal to the argument x.

Error Checks

A number of error checks have been built into the FORTRAN subroutines. The basic philosophy that has been followed with respect to an error situation is to have an error message typed out, to set the result of the operation equal to the most reasonable value under the circumstances, and to have the program continue. The following list shows the error checks that exist in the subroutines, the error codes that are typed out, and the value to which FAC is set before the program continues. In the list it will be noted that the terms "Overflow" and "Underflow" occur several times. Overflow means that the characteristic of the result has exceeded 99; underflow means that the characteristic of the result is less than —99.

ERROR CHECK	ERROR CODE	CONTENTS OF FAC
Overflow in FAD or FSB	E1	$\bar{9}99\ 999\ 99\bar{9}9$
Underflow in FAD or FSB	E2	$\bar{0}00\ 000\ 00\bar{9}9$
Overflow in FMP	E3	$\bar{9}99\ 999\ 99\bar{9}9$
Underflow in FMP	E4	$\bar{0}00\ 000\ 00\bar{9}9$
Overflow in FDV or FDVR	E5	$\bar{9}99\ 999\ 99\bar{9}9$
Underflow in FDV or FDVR	E6	$\bar{0}00\ 000\ 00\bar{9}9$
Zero divisor in FDV or FDVR	E7	$\bar{9}99\ 999\ 99\bar{9}9$
Zero divisor in FXD or FXDR	E8	999 999 9999
*Argument in FIX $\geq$ 10,000	E9	$\bar{9}99\ 999\ 99\bar{9}9$
*Argument in FIX $\leq$ -10,000	E9	$\bar{9}99\ 999\ 9\bar{9}9\bar{9}$
Loss of all significance in FSIN or FCOS	F1	$\bar{9}99\ 999\ 99\bar{9}9$
Zero argument in FLN	F2	$\bar{9}99\ 999\ 9\bar{9}9\bar{9}$
Negative argument in FLN	F3	$\pounds n\,\lvert x\rvert$
Overflow in FEXP or FEXN	F4	$\bar{9}99\ 999\ 99\bar{9}9$
Underflow in FEXP or FEXN	F5	$\bar{0}00\ 000\ 00\bar{9}9$
Negative argument in FAXB	F6	$\lvert A\rvert^{B}$
Negative argument in FSQR	F6	$\sqrt{\lvert x\rvert}$
Input data in incorrect form or outside allowable range	F7	
Floating point output data outside allowable range, or in form not acceptable to FORMAT specification	F8	
Input or output card record is longer than 72 characters, or there is an element in an input or output list for which there is no specification in the corresponding FORMAT statement	F9	

* Floating-point hardware subroutine only

Input/Output Data. Input data to the object program is read alphamerically at the paper tape reader, the card reader, or the console typewriter.

If error F7 occurs during the execution of the instructions compiled for an input statement, the data which is incorrect will be ignored and processing will continue.

If error F8 occurs, the incorrect data will be ignored in the output record, and an additional record will be created containing the incorrect data in the form specified by E14.8 for floating point data. Fixed point data outside the range of the format specifications will be output in the form I (w-1) where w is the specified width. No error indication will occur.

If error F9 occurs, the incorrect data will be ignored and processing will continue.

Adding Subroutines

As indicated earlier, up to 19 additional subroutines can be added to the 31 subroutines provided by the program. Additions of relocatable subroutines to the FORTRAN system involve changes in the language, the processor, and the subroutines.

Language

The four type C subroutines provided with the system may be replaced subject to the restrictions mentioned below. The two type B subroutines are an integral part of the system and may not be replaced.

Subroutines added to the system are type C. Such subroutines must be given a one-to-four character symbolic name. For example, a subroutine to calculate hyperbolic sine might be called SNH, and in a source program might be used in such a statement as

$$Y = SNH(X) \text{ or as } Y = SNHF(X).$$

Processor

A 4-character record in the processor specifies the number of functional subroutines included in the subroutine deck or tape to which direct reference may be made in a source program. There are six such subroutines included in the standard system, and the record bb06, where b is a blank, is punched in the processor. This record appears in card number 02001 in the processor deck and is the fifty-sixth record in the processor tape. If additional subroutines are added, or if some of the available subroutines are not used, this record must be changed to the actual number of type B and type C subroutines included in the program. Immediately following the 4-digit record specifying the number of included functional subroutines, are cards in the processor deck or records in the processor tape giving the symbolic names of the associated functions. Each symbolic name must be preceded by two blanks. In the card system, a 5-digit sequence number must appear in columns 76 through 80, starting with 02002. The names of the subroutines in the standard system and the order in which they appear are as follows:

Subroutine Name	Subroutine Number	
SIN, COS	4, 5	(treated as one subroutine)
ATAN	6	
EXP	7	
LOG	8	
SQRT	9	

This is an ordered list, and the sequence in which the function names are read by the compiler must not be changed. Each subroutine, starting with SIN is assigned a serial number NN, dependent upon its position in the list, to which the subroutine relocator program refers. The serial number of the SQRT routine, for example, is 09. The addition of the hyperbolic sine routine mentioned above would require the addition of a card numbered 02008 containing the name SNH punched in card columns 3 through 5. The serial number automatically assigned to this function would be 10. Subroutine numbers 1 through 3 cannot be used.

Subroutines

The subroutine relocator routine contained in the subroutine deck or tape will relocate and reproduce into the object program, or store in core storage, any relocatable subroutines called for by the source program. The first relocatable subroutine will start in the next even address beyond the object program.

The relocator requires the relocatable subroutines to be in the same order in the subroutine deck or tape as their corresponding symbolic names appear in the processor. In addition, there must be a relocatable subroutine in the subroutine deck or tape for each symbolic name used in the processor.

All relocatable subroutines have been written in 1620 SPS language. In the card system, the assembled object programs have been condensed by the SPS condensing routine; the first two and last seven cards of the condensed output have then been discarded. In the tape system, the 2-record loading routine at the beginning and the single record containing the multiplication and addition tables at the end of the SPS output are removed. A flag is inserted in the low-order position of the 10-digit loader record that precedes *instructions* only (XXXXX XXXX$\overline{X}$). The header and trailer records are added, and in the card system proper sequence numbers are punched in colums 76 through 80.

For the paper tape system, an SPS modification tape is included which will modify the standard 1620 SPS paper tape system (1620-SP-008) so that the header and trailer records will automatically appear in the SPS output tape. To use this

tape after the SPS system has been loaded in the normal way, insert the instruction 36 00000 00300, mount the modification tape, and press the release and start keys. After the modification tape has been read in, the following message will be typed:

TYPE IN TWO DIGIT SUBROUTINE NUMBER

The correct 2-digit subroutine number must be typed, and the release and start keys pressed. If a typing error is made, the error may be corrected by using program switch 4, as described under OPERATING PRINCIPLES. Processing may be continued by entering the SPS source program.

Since the sine-cosine subroutines are together as one subroutine with different entries, they must remain in the subroutines for compatibility with the relocator. However, if you wish to write a new sine-cosine subroutine, it must be compatible with the relocator, i.e., the sine entry equals cosine entry +44.

Writing Relocatable Subroutines in SPS

The origin of a relocatable subroutine must be at location 5000, and must be the address of the first instruction executed in the subroutine. Relative addresses in an instruction are indicated by flags over the 0 or 1 positions of the operation code. For example, if the P address of an instruction is relative to the origin 5000, a flag must be over position 0. The P address will then be modified when the subroutine is relocated. The flags are not removed by the relocator but are stored in memory with the instruction at object time. Since relative P and Q addresses are to be modified, they must not contain any flags other than in the P_1 or Q_7 positions. (Flags on P_1 or Q_7 are not necessary to the subroutine relocator.)

The address of the argument will be found in location 19989 −20(NN −4) where NN is the subroutine number. (If compilation needs additional memory, the location of the argument must be modified by 20 or 40 K, depending upon the amount of additional memory used.) The calculated result of a relocatable subroutine must be left in the floating point accumulator (FAC, 00051 through 00060), or a flag must be set in location 00051. Although record marks may be contained within a subroutine, one is available in location 00401).

Relocatable subroutines must exit by a Branch Back (BB).

A flagged digit, representing the high-order digit of the highest numbered core storage location used in the system, is in location 00400. This digit is, for example, $\bar{5}$ for a 60,000 location machine configuration.

Header Record

In the card system, the header card has the following form:

Columns	1-2	Subroutine number (NN).
	62	Zero.
	76-80	Sequence number (sequence number = NN000, where NN is the subroutine number. The first sequence number in the subroutine itself would then be NN001.)

In the tape system, there are two header records: the first contains a single zero and the second contains the 2-digit subroutine number.

Trailer Record

In the card system, the trailer card has the following form:

Columns	1-5	The next even number above the number of locations used by the subroutine.
	62	$\bar{0}$ (flag zero).
	76-80	Sequence number.

In the tape system, the first two of the preceding items are reversed and appear as individual records, i.e., the first record contains a flag zero and the second record contains the next even number above the number of locations used by the subroutine. The last card of the relocatable subroutine section of the subroutine deck contains a flag one (1̄) in column 62 and the sequence number 29000 in columns 76 through 80. This card follows the last trailer card and indicates to the relocator that all relocatable subroutines have been processed. In the paper tape system, this record is a single one (1).

Writing Relocatable Subroutines in Machine Language

If a relocatable subroutine is written in machine language, the origin and operation code flags must be as described for writing in SPS. The card format must also conform to the condensed SPS as follows:

Instruction Card

Columns 1-61	One to five instructions with operation codes flagged for relative P or Q addresses. A record mark must be in column 61 or must immediately follow the last instruction on the card (the record mark is not loaded at object time). Instructions must use the full 12 digits. If packing is done, the Q field must still be filled with zeros and the packed instructions would start a new card.
62	0 (zero-instruction card).
65-69	Storage address where column 1 of the card will load. **(High-order digit must be flagged.)**
70-74	Address of next storage location beyond the number of locations used by the instruction. **(High-order digit must be flagged.)**
76-80	Sequence number.

Constant Card

Columns 1-61	Constants which will be loaded sequentially into memory. A record mark must be in column 61 or immediately following the last digit of a constant on the card. Consecutive constants terminated by record marks must be on individual cards with double record marks at the end.
62	1 (one-constant card).
65-69	Same as instruction card.
70-74	Address of next core location beyond number of locations used by constants. **(High-order digit must be flagged.)**
76-80	Sequence number.

In the paper tape system, an absolute language version of a relocatable subroutine must be in the same form as output by the paper tape version of 1620 SPS.

In the card system, the subroutine relocator checks sequence numbers upon reading. If a card is missing or out of order, the error message

CARD OUT OF SEQUENCE

will be typed. In this case you must restore the proper sequence and then push the start key.

Storage Allocation

After Loading the Compiler

After the standard processor deck or tape has been read into 1620 storage, and before processing of the source statements has begun, storage is allocated as follows:

1. The multiply-add tables are in locations 00100 through 00399.
2. The compiler program begins in location 00402.
3. The work areas in which the source program will be processed have been cleared where necessary. These areas, and a constant defining the end of the symbol table area used for function names, are in locations 16800 through 17498.
4. In the standard system (20,000 positions of storage), twelve 10-digit fields are located in positions 19880 through 19999. The alphabetic representation of the names of the six relocatable subroutines, in the two forms allowed – one with and one without the terminal F – are stored in the 12 fields. The name of each additional relocatable subroutine inserted by the user will be added to this list, and will appear in the symbol table in both forms.
5. The rest of the symbol table from location 17500 through 19879 contains 238 10-digit fields, each containing the constant 00000000 ≠ ≠. The end of the symbol table is defined by the constant 0 ≠ in locations 17498 through 17499.

If the system has been modified for use with the 1623 Core Storage unit, the symbol table will occupy the highest positions of storage. If, for example, the highest available address is 59999, the subroutine names will appear in locations 59880 through 59999. The constant defining the end of the symbol table will be in locations 40008 through 40009.

After Processing the Source Program

After compilation, the areas previously cleared for the symbol table will contain:

1. The alphabetic form of every variable used in the source program.
2. In the next lower field after the name of every variable array, a field of the form $\overline{0}\,\overline{1}$ I I I $\overline{\text{N}}$NNNN, where the Is represent the first specification listed in the DIMENSION statement for the array, and the Ns represent the address of the last element in the array.
3. Every constant used in the source program. Floating point constants will have the form of an 8-digit mantissa and a 2-digit exponent. Fixed point constants are in 4-digit subfields (right-justified) within the 10-digit fields in which they appear. All constants have a flag over the low-order digit.
4. All statement numbers will be in the form $\overline{\text{L}}$LLLLL0$\overline{\text{M}}$MMM, where the Ms represent the statement number, and the Ls the location in the object program of the first instruction compiled for the source statement indicated.
5. Intermediate storage, or accumulator numbers, from $\overline{0}$00 through $\overline{9}$98, as required and assigned by the compiler.
6. In the next lower field after the final field used in the symbol table by the compiler, the constant $\overline{0}$000000$\overline{9}$99 will appear.

A record, consisting of three 5-digit fields which has been punched into the object program at the conclusion of compilation, is stored in locations 00402 through 00416. The first of these fields contains the address of the first location available for the storage of relocatable subroutines after the object program has been properly loaded. The next field contains the address of the end of the symbol table when it is loaded at object time, and the third field contains the corresponding address for the symbol table as it appears in compressed form at the end of compilation.

A 50-digit record is in location 00418 through 00467, which indicates the particular relocatable subroutines to be added to the object program by the subroutine relocator program. The digit 1, appearing in an odd position of this record, reading from right to left, is interpreted as meaning that the corresponding subroutine is to be included (the 6 relocatable subroutines, and then the 19 optional subroutines). The order of the indicators is the same as the order in which the names of the subroutines are read in during the initialization phase.

After Loading the Object Program

After the object program has been loaded, including the subroutines, if necessary, the multiply-add tables are in locations 00100 through 00399. The arithmetic and input/output subroutines, together with the work areas they require, begin in location 00402. The object program begins at location 08300 and is followed by any relocatable subroutines called for by the source program. The symbol table has been loaded and modified to form a data table. Locations 00051 through 00099 are used for intermediate storage and a product area required by multiply instructions. The following illustration shows the location in storage of all subroutines except FSIN, FCOS, and FATN.

Location of Subroutines at Object Time

Subroutine	Symbolic Name	Storage Location
Floating Add	FAD	00518
Floating Subtract	FSB	00408
Floating Multiply	FMP	01378
Floating Divide	FDV	01862
Reverse Floating Divide	FDVR	01756
Floating A**B	FAXB	03270
Floating A**B(-B)	FAXBN	04232
Fixed Add	FXA	02644
Fixed Subtract	FXS	02700
Fixed Multiply	FXM	02748
Fixed Divide	FXD	02876
Load Into FAC	TOFAC	01238
Store from FAC	FMFAC	01306
Reverse Fixed Divide	FXDR	02816
A**I	FAXI	03670
A**(-I)	FAXIN	03622
Convert Sign	RSGN	02546
Convert:float-to-fix	FIX	03494
Convert:fix-to-float	FLOAT	03222
Read Card	RACD	04512
Read Tape	RAPT	04596
Read Typewriter	RATY	04548
Write Card	WACD	04748
Write Tape	WAPT	04844
Write Typewriter	WATY	04796
Trace	TRACE	05124

System Deck

General Make-up of the Compiler Deck

The compiler deck is comprised of two programs separated by a group of cards consisting of an object program loader and the number and names of the subroutines included. The first program is the initialization phase which reads in the object program loader and punches it out into the object deck. This program continues by initializing the symbol table area and the area into which source statements will be read. Finally, a card containing the number of subroutines included, and individual cards containing the names of the subroutines are read in and processed. The second of the two compiler programs is then read in and

a halt instruction is executed. The starting instruction for each program is in location 00402. After each statement is processed during compilation, the program returns to location 00462 to continue.

The sections of the standard deck, identified by card number, are as follows:

Card Numbers	
00001 through 00044	Loading routine and initialization program
01001 through 01054	Object program loader
02001	Number of included subroutines
02002 through 02007	Names of included subroutines
03001 through 03229	Compiler program

General Make-up of the Subroutine Deck

The first section of the subroutine deck is a loading routine which loads the subroutine relocator. The relocator processes the relocatable subroutines which immediately follow it in the deck and finally reads in and processes the arithmetic and input/output subroutines (type A) which are contained in the last section of the deck.

The sections of the standard subroutine deck, identified by card number, are as follows:

Card Numbers	
04001 through 04008	Loading routine
04009 through 04059	Subroutine relocator
05000 through 05019	SIN/COS subroutine
06000 through 06029	ATAN subroutine
07000 through 07015	EXP subroutine
08000 through 08022	LOG subroutine
09000 through 09017	SQRT subroutine
29000	Relocatable subroutine trailer
30000 through last card	Arithmetic and input/output subroutines

General Make-up of an Object Deck

The first two sections in the object deck have been punched during the initialization phase and consist of a short loading routine which loads the add tables and the program and symbol table loader. The cards following these contain the compiled instructions which are concluded by a record containing only the constant $\bar{0}\bar{0}09990$ and a communication card. The communication card consists of three 5-digit fields, the 50-digit field indicating which subroutines are being used, followed by a 5-digit field indicating the memory capacity. When executed, the first loading routine branches to the program loader which loads the compiled instructions in proper order into storage to form the object program. Following this is the symbol table, as it appears at the end of compilation, which is read into storage by the program loader. These cards are read into storage by the program loader which expands the table to allow for any dimensioned variables which were used in the source program. The next section contains any relocated subroutines (type C) which may have been called for, if the subroutines were processed when the object program was compiled.

The last section of the deck contains the arithmetic and input/output subroutines, the multiply and add tables, and the instructions which cause the machine to halt before branching to the start of the object program.

The sections of the object deck, identified by card number, are as follows:

Card Numbers	
0001 through 0008	Loading routine and add tables
0009 through 0054	Program loader
0055 through last card	Compiled instructions
	Communication card
	Symbol table
	Relocated subroutines and arithmetic and input/output subroutines, when required.

NOTE: When the symbol table is loaded, only constants and statements are placed into the data tables.

Variables computed in a FORTRAN object program are stored in specific 10-digit fields in core storage, the addresses of which have been determined in the compilation process. Addresses are assigned in descending order from the highest-numbered storage location, in the order in which the variables, constants, and statement numbers are encountered in the source program. The order of address assignment is repeated for each object program compiled. The values computed and stored during the execution of an object program are not disturbed by the loading of another object program, if the variables have appeared in the second source program in **exactly** the same order as in the first. By this means, for example, if an array of variables is computed in an object program, another object program may be loaded immediately to use the same computed values in further computations. The names of the variables used in this way need not be the same from one source program to another.

An involved algebraic calculation might require the use of temporary storage fields which are automatically assigned by the compiler. For this reason, variables **appearing** in the same order, but which are defined for the first time in the body of different source programs may not be given the same assigned address. Symbol table listings at compilation time will disclose any such lack of correspondence.

System Tapes

General Make-up of the Compiler Tape

The compiler tape consists of two programs separated by a group of records which are processed when the tape is read into the 1620. The first program is the initialization phase which reads in the first five records on the tape following the program itself, and punches them out into the object tape. This program then initializes the symbol table area and the area into which source statements will be read. Finally, a record containing the number of the subroutines included and individual records containing the names of the subroutines are read in and processed. The rest of the tape which contains the compiler is then read in and a halt instruction is executed. The starting instruction for each program is in location 00402. During compilation, after each statement is processed, the program returns to location 00462 to continue.

General Make-up of the Subroutine Tape

The first section in the subroutine tape is a loading subroutine which causes the subroutine relocator program which follows it to be read and executed. The relocator processes the relocatable subroutines which immediately follow it on the tape, and finally reads in and processes the arithmetic and input/output subroutines which are contained in the last record on the tape.

General Make-up of an Object Tape

The first five records in the object tape have been punched during the initialization phase, and contain a short loading routine which loads the multiply-add tables and the program loader. The records following are the compiled instructions which are concluded by a record containing only the constant $\bar{0}0\bar{0}09990$. The first loading routine branches to the program loader which loads the compiled instructions in proper order into storage to form the object program. Immediately following the compiled instructions is a record consisting of three 5-digit fields and a 50-digit field that indicates which subroutines are being used, followed by a 5-digit field that indicates the memory capacity. The symbol table follows (punched in 60-character records) as it appears at the end of compilation, and is read into storage by the program loader. The symbol table is expanded as it is loaded to allow for any dimensioned variables which were used in the source program. The next section contains any relocatable subroutines (type C) which may have been called for when the object program was produced. The records following the relocatable subroutines modify a loading routine to read in a record containing the arithmetic and input/output subroutines (type A). The last records in the object tape modify the routine to cause it to read in and type out the message which calls for the loading of data, and to come to a halt before starting the execution of the object program. See the note at the end of the description of the general make-up of the object deck.

Making Corrections to FORTRAN System Tapes

The loading routine used to read in the compiler program requires two records to load information into storage. The first record is in the form

$$\bar{L}LLLL\ \bar{H}HHHH$$

where the Ls represent the low position into which the data is to be read, and the Hs represent the location immediately following the last location to be used. The records following are read into storage in accordance with the addresses given. Corrections to the compiler program are prepared in the form required by the compiler loading routine, punched in paper tape, and may be inserted in the compiler tape by using the following method.

1. Duplicate the processor tape by means of the special duplicating program, then single instruction execute the machine toward the end of the original tape until the third record from the end has been read into storage. Remove the processor tape from the tape reader.
2. Mount the correction tape and continue duplication until the last record has been read, then remove it from the reader.
3. Replace the processor tape at the start of the second record from the end and complete the duplicating process.

The duplicated tape will contain the new information desired, and will cause the machine to execute the normal halt immediately after loading.

Duplicating the Processor and Subroutine Tapes

Description of the Program

The purpose of this program is to duplicate the FORTRAN processor and subroutine tapes for use on the basic 1620 system (20,000 storage positions), or to duplicate and alter the processor tape for use on 1620 systems that utilize additional memory (40,000 or 60,000 storage positions). Program switches 1 and 2 control the setup for the tape to be duplicated and also control whether alterations are to be made to the tape. Since the processor tape contains some alphabetic records, a special test is incorporated in the duplicating program to

reproduce these records as well as the numerical records. The duplication of the subroutine tape is entirely numerical.

Restrictions to this program are as follows:

1. The maximum permissible record length is 9000.
2. This program is intended to be used to duplicate FORTRAN processor and subroutine tapes only.

Operating Procedure

The procedure for using the program is:

1. Thread the processor and subroutine duplicating tape.
2. Set the parity and I/O switches to STOP, set the MAR switch, if any, to STOP, and the OFLOW switch to PROGRAM.
3. Press the reset and insert keys.
4. Insert 36 00000 00300.
5. Press the release and start keys to load the duplicating program.
6. Run out the duplicating tape and thread the FORTRAN tape to be duplicated (processor or subroutine).
7. Set the parity and I/O switches to PROGRAM, the MAR switch, if any, to STOP and the OFLOW switch to PROGRAM.
   a. For duplicating the processor tape, set program switch 1 ON, and program switch 2 OFF.
   b. For duplicating the subroutine tape, set program switch 1 OFF and program switch 2 OFF.
   c. For modifying the processor tape, for 40,000 or 60,000 storage positions, set program switches 1 and 2 ON.
8. Ready the punch.
9. Press the start key.
   a. If program switch 1 is ON or OFF and 2 is OFF, the tape duplication will begin.
   b. If program switches 1 and 2 are ON, the following message will be typed after twelve records have been duplicated:

      TYPE SIZE OF MEMORY IN THOUSANDS

      After typing the specified information, press the release and start keys and the tape duplication will continue.

      If you have made an error in typing , you may recover in the following manner: turn switch 4 ON, press release and start keys, turn switch 4 OFF, re-enter the information. This process may be repeated.

To duplicate another tape (processor or subroutine), thread the tape, ready the punch, press reset and insert, insert 49 00966, and press release and start keys. **Make certain that program switches are set correctly each time you repeat the duplication process.**

Error Detection

During the duplication, one or two error messages may be typed out after which the machine will halt.

1. "ERROR 1"— An invalid character has entered the input area. Back the tape up to the beginning of the record and press the start key. If the error message is repeated, examine the tape for an invalid character.
2. "ERROR 2"— A machine error has occurred. Back the tape up to the beginning of the record and press the start key. If the program hangs up or keeps typing ERROR 2 messages, this indicates that a portion of the program may have been destroyed. If this occurs, reload the program and restart the duplication process.

Tape Duplication

The procedure for duplicating the "1620 Program for Duplicating the FORTRAN Processor and Subroutine Tape" is:

1. Thread the FORTRAN processor and subroutine duplicating tape.
2. Set the parity, I/O, MAR (if any), and OFLOW switch to STOP.
3. Ready the punch.
4. Press the reset and insert keys.
5. Insert 36 00500 00300
   38 00500 00200
   49 00000
6. Press the release and start keys.

To duplicate the 1620 FORTRAN-SPS Modification tape:

1. Thread the FORTRAN-SPS Modification tape.
2. Proceed as in step 2 above.

Modification of 1620 FORTRAN for Additional Core Storage

The standard FORTRAN system decks and tapes, as issued, do not require a machine system containing more than 20,000 positions of core storage. The processor tape or deck must be modified to allow the use of the 1623 Core Storage Unit. No modification of the subroutines is necessary.

Modifying the Processor

In the card system, the two high-order digits of the highest address in storage are punched in card columns 25 and 26, or card number 00025 in the processor deck. To modify the program for additional storage, duplicate card 00025 with the proper digit in column 25. If, for example, the deck is to be used with a system in which the highest address is 39999, punch into column 25 the flagged digit $\bar{3}$.

To modify the tape system, use the 1620 program "Duplicating the Processor and Subroutine Tapes."

Part 5—The FORTRAN Pre-Compiler Program

The IBM FORTRAN Pre-Compiler is a program that detects and permits correction of errors in a FORTRAN source program before the object program is compiled. The Pre-Compiler detects many of the more common programming errors in individual source statements, and indicates possible logical errors in the source program as a whole.

Two versions of the Pre-Compiler are provided, one for use with the IBM 1621 Paper Tape Reader and IBM 1624 Tape Punch, and the other for the IBM 1622 Card Read Punch. A FORTRAN source program which is to be processed may be punched in paper tape or cards, or may be entered directly from the typewriter.

The operation of the Pre-Compiler can be divided into two phases: Error Analysis and Final Program Summary.

During the error analysis phase, each statement in the FORTRAN program is analyzed for an error. If an error is detected, an error code is typed, the statement containing the error is typed, and the program halts so that you can type the statement correctly. During this phase, a new FORTRAN source program can be punched in paper tape or cards. After all statements have been analyzed and corrected, if desired, a final program summary is typed.

The final program summary phase includes information about possible sources of errors not detectable in individual source statements.

An additional feature of the Pre-Compiler program, using the 1620 program switches, permits you to easily alter the functions of the error analysis phase for individual requirements. The following options are available.

1. You can suspend the halt and error correction routines, thereby providing a quick error analysis only. These routines can be suspended for the entire program or for individual error halts during normal processing.
2. You can enter the program through the console typewriter rather than by card or paper tape input.
3. You can eliminate punching of an edited source program.
4. You can have correct program statements typed, in addition to the normal operation in which only incorrect statements are typed.

The standard 1620 FORTRAN Pre-Compiler system contains all of the functional subroutine names included in the standard FORTRAN system. If you make alterations to the functional names, or if you add additional subroutines to the FORTRAN system, you must make the corresponding alterations and additions to the Pre-Compiler system.

Operation of the Pre-Compiler Program

Before you process any program on the Pre-Compiler, you should be familiar with the nature of the errors that the program is designed to detect.

Error Codes

During the error analysis phase of the program, each statement is analyzed for one or more specific errors. These errors consist of 51 of the most common errors usually found in FORTRAN source programs. As an aid in evaluating these errors, they have been grouped into seven categories:

Arithmetic statements

Variables in arithmetic statements
DO loops
Constants
Statement numbers
Transfer statements
General

When an error is detected, an error code is typed on the console typewriter. This code consists of an alphabetic abbreviation of one of the categories listed above, followed by a number that designates the particular error in the category.

Arithmetic Statements

ARITH

1. Unacceptable form to left of = sign.
2. Multiple = signs.
3. This code has been deleted.
4. Successive operation symbols, or a function which is followed by an operation symbol.
5. Missing operation symbol or operand.
6. Right parenthesis encountered before corresponding left parenthesis.
7. Missing right parenthesis.
8. Mixed mode expression (expression contains fixed and floating point).
9. No variable to the left of equal sign.
10. Involution of a fixed point variable or constant.

Variables in Arithmetic Expressions

VAR

1. Variable name longer than 5 alphameric characters.
2. Variable appearing in an expression or as a subscript not previously defined in an input statement; as the index of a DO loop; or defined as the left side of another arithmetic statement.
3. Variable written with a subscript has not been previously defined in a DIMENSION statement.
4. Variable previously defined in a DIMENSION statement has not been subscripted correctly: subscript is in unacceptable form, number of subscripts does not agree with the number specified in DIMENSION statement, numerical subscript is greater than maximum allowed by DIMENSION statement or is less than 1.

DO Loops

DO

1. In the statement DO n $i = m_1, m_2, m_3$, the indices m_1, m_2, and m_3, if given, are not all unsigned fixed point variables or constants greater than zero. There are more than 3 indices given.
2. The second index, m_2, is less than m_1, when both are constants.
3. The third index, m_3, is signed, is zero, or is missing when specified as a constant.
4. The statement number n is not in acceptable form or is missing.
5. The variable name has either been omitted, or is incorrectly stated, or the DO statement is incorrect.
6. The statement specified as the end of an outer loop in a nest of DO's has been found before an inner loop is complete.
7. A DO loop terminates with a transfer statement, GO TO, computed GO TO, or IF.

Constants

CONST

1. Fixed point constant longer than 4 digits.
2. Floating point constant outside the allowable range.
3. Decimal point omitted from floating point constant that is written with a decimal exponent.

4. The decimal exponent following the E in a floating point constant is incorrectly expressed in form or size.
5. The exponent following an E has been omitted.
6. Floating point number followed by an alphabetic character other than E.

Statement Numbers

STNO

1. Statement number longer than 4 digits.
2. Statement number has been previously defined.
3. Unnumbered CONTINUE statement. (Should be numbered when used as last statement in a DO loop.)
4. Statement immediately following a transfer statement is not numbered, and is therefore inaccessible to the source program. (If the previous statement is a transfer, the only way the program can process this statement is by a transfer to it, and therefore it must always be numbered.)

Transfer Statements

TRANS

1. Statement numbers in a transfer statement (GO TO, computed GO TO, or IF) are not acceptable fixed point numbers; there is no comma between statement numbers, or there is not the required number of statement numbers.
2. Comma missing after the right parenthesis in a computed GO TO statement.
3. Index in a computed GO TO statement is not a fixed point variable, or is missing.
4. Nonnumerical character follows right parenthesis in an IF statement.
5. In an IF statement, a character other than a left parenthesis follows the word IF.
6. No arithmetic statement within the parentheses after the IF. (However, empty parentheses in an arithmetic statement will not be detected.)

General

GEN

1. Misspelled or unacceptable nonarithmetic statement.
2. Statement contains an unacceptable character.
3. More than 72 characters in statement (not applicable to cards).
4. Symbol table full (occupies more than 2,500 digits in storage).
5. Statement contains decimal point that is not in a floating point constant.
6. Input/output statement contains no FORMAT number, or is incorrectly stated.
7. First character in an input/ouput list is not alphabetic, or the final character is not a letter or a digit.
8. In a DIMENSION statement, a nonalphabetic character precedes the first variable name or a dimension, or three dimensions have been specified (only two-dimensional arrays are permitted).
9. A specified dimension is incorrect: a parenthesis has been omitted, a floating point constant or an unacceptable fixed point constant has been used, etc.
10. Unnumbered FORMAT statement.
11. Incomplete FORMAT statement: invalid or incorrect specification, missing parentheses, character after right parenthesis, etc.
12. In an input/output statement, comma is missing after the FORMAT statement number, or the list is missing or invalid.
13. The total record width specified in a FORMAT statement exceeds 87.
14. A variable appearing in a DIMENSION statement has been previously defined.
15. H or X missing in alphameric FORMAT specification or the width of alphameric specification is greater than 49.

Error Analysis Phase

After an error has been deteced in a statement, and the appropriate error code has been typed, the original error statement is typed. If switch 3 is off, the carriage is returned and the program halts to wait for a corrected statement to be typed in. After reviewing the erroneous statement and the error code or codes indicated, you can, in most cases, make an immediate correction to the statement. Type the correct statement (followed by a record mark), then press the release and start keys. The program resumes by analyzing the statement just typed to determine if any errors still exist. If the statement is correct, the program will begin analyzing the next statement in the FORTRAN program.

In some cases, it may not be possible to re-enter a corrected statement without certain modifications because part of the statement has already been processed as a correct statement. For example, if an error is discovered in a transfer statement (GO TO), you must enter the correct statement with a statement number to avoid error STNO 4, or enter it twice without a statement number. (The program considers the first part of the GO TO to be correct, and requires that any statement following a transfer statement must contain a statement number).

When a statement containing a statement number is partially processed due to an error, you cannot re-enter the statement with the statement number because an error STNO 2 will result.

In an erroneous DIMENSION statement, for example DIMENSION C(N), the C is stored as a nonsubscripted variable and cannot be used later in the program as a subscripted variable. In case of C(10,N), the C is stored as a one-dimensioned variable. Restart of the Pre-Compiler is necessary.

For expressions involving involution (raising to a power), the exponent cannot have an involution operation. For example, A**(B**2.+.1) will result in erroneous operation of the Pre-Compiler. This restriction also applies to the arithmetic expression in an IF statement.

There is no check for the termination of a DO loop. If a dimension specification exceeds the capacity of the storage, erroneous results will follow.

If an immediate correction cannot be made, you can resume testing of the next statement by manually branching to BEGIN (see RESTART PROCEDURES).

It is important to note that if a new source program is being punched, bypassing the error correction routine will result in the incorrect statement being punched into the output tape or cards.

The normal operation of the Pre-Compiler program is to type incorrect statements only. If you require a typed copy of all statements, turn on program switch 1.

Restart Procedure

You may find it necessary during processing to interrupt the normal operation of the program. To allow such interruptions, the following re-entry points, given by symbolic label and storage location, are available:

CLEAR location 01208: The symbol table and table of statement numbers referenced by DO statements are cleared. CLEAR is the restart point for a new program to be tested.

INITL location 01340: The table of statement numbers referenced by DO statements is cleared.

BEGIN location 01472: No tables are cleared, but the program will continue to read source program statements. BEGIN is the normal entry point for restarting after a check stop or other interruptions of the Pre-Compiler.

Final Program Summary

After the END statement in a source program has been processed by the Pre-Compiler program, a final program summary is typed on the console typewriter.

The summary includes information about possible sources of error not detectable in individual source statements, and is in the form of four alphabetic messages together with related lists, as follows:

UNDEFINED STATEMENT NUMBERS
$\overline{n}\ n\ n\ n$
$\overline{n}\ n\ n\ n\ldots$

The numbers listed are those which have not been used for statement identification but have been referenced by transfer or DO statements.

UNREFERENCED STATEMENT NUMBERS
$\overline{n}\ n\ n\ n$
$\overline{n}\ n\ n\ n\ldots$

The numbers listed are those which have been used for statement identification but have not been referenced by transfer or DO statements. These numbers are not necessary to the compilation of the source program and may be eliminated.

RELOCATABLE SUBROUTINES CALLED
LOG
SIN....
:

The names listed are those of the functional subroutines used in the source program.

OBJECT PROGRAM DATA TABLE
XXXXX STORAGE POSITIONS

The number of storage positions given includes those used for variables, constants, and statement numbers, but not the total number of storage positions that will be required in the FORTRAN object program, since this depends upon the number of machine instructions produced when the source program is compiled.

Premature typing of the summary indicates that the END statement appears earlier than anticipated in the source program. Conversely, if the END statement has been omitted, the summary will not be typed.

If statement number 999 is used it will cause errors in the final program summary. However, no damage will be done to the Pre-Compiler.

Interpretation of Detected Errors

An expression may appear so ambiguous to the Pre-Compiler program that any of several possible errors might be detected. For example in the expression

ABE(C+D)

if ABE is not the name of a function, and has not been defined previously in the program, it might be regarded as a subscripted variable name with subscripts written in an unacceptable form. When the name ABE has been defined as a non-subscripted variable, however, the obvious error is that of omission of an operation symbol.

Under certain conditions, an error in one source statement may affect the validity of other statements which follow it in the source program. It is recommended that a new FORTRAN source program tape or deck produced by means of the Pre-Compiler program be reprocessed until no errors can be detected.

Conditions which might possibly lead to error have been assigned error codes or are noted in the final summary. The fact that a statement is indicated

to be in error does not necessarily mean that the source program cannot be compiled correctly or that the object program cannot be successfully run. Conversely, a source program which has been processed by the Pre-Compiler and found free of error might have certain undetectable mistakes in logic, or be too large for the particular 1620 system in use.

The 1620 FORTRAN Pre-Compiler cannot determine the **intent** of your program. Even though no errors are present in individual source statements, you should examine the final program summary to determine if any logical errors in the flow of the source program still remain to be corrected.

Program Switch Settings

The possible settings for the program switches are shown below

Input	Print On Typewriter	Punch Edited Source Program	SW 1	SW 2	SW 4
Cards/Tape	Yes	Yes	On	On	On
Cards/Tape	Yes	No	On	Off	On
Cards/Tape	No	Yes	On	On	Off
Cards/Tape	No	No	On	Off	Off
Typewriter	No	Yes	Off	On	On/Off
Typewriter	No	No	Off	Off	On/Off

Switch 3 has the following function:

On – Error correction routines are bypassed.

Off – Error correction routines are not bypassed.

NOTE: Switch 4 is normally turned off. When you make an error in typing either an original or a corrected source statement, turn this switch on, press the release and start keys, and return the switch to its normal off position. You must then retype the entire statement.

Processing with the Pre-Compiler Program

Loading the Program — Card Deck

The sequence of operations required to load the program card deck is as follows:

1. Clear core storage to zeros. A suggested method for clearing to zero is to:
   a. Press the reset key.
   b. Press the insert key.
   c. Type the instruction 16 00010 00000.
   d. Press the release key.
   e. Press the start key.
   f. When all storage position have been cleared, press the instant stop key.
2. Set the console program switches for the input/output option you want.
3. Set the overflow check switch to PROGRAM and all other check switches to STOP.
4. Press the reset key.
5. Place the deck in the read hopper and press the load key.

The cards comprising the FORTRAN Pre-Compiler deck are punched with sequence numbers in columns 76 through 80 and the deck must be loaded in sequence.

Loading the Program — Tape File

The sequence of operations required to load the program tape is as follows:

1. Clear core storage to zeros, set the console program, and check switches as in steps 1, 2, and 3, just given.

2. Mount the program tape.
3. Press the reset key.
4. Press the insert key.
5. Type the instruction 36 00000 00300.
6. Press the release key.
7. Press the start key.

Processing the Source Program

After the Pre-Compiler has been successfully loaded, the following instructions will be typed on the console typewriter:

ENTER SOURCE PROGRAM
THEN PUSH START

and the program will halt. Set the console program switches for the correct input/output options, mount the source tape or load the source deck, and begin processing by pressing the start key.

After the END statement in a source program has been processed, the final program summary will be typed on the console typewriter. This will consist of the following four messages, together with the related lists of statement numbers and relocatable subroutines called by name in the source program.

UNDEFINED STATEMENT NUMBERS
UNREFERENCED STATEMENT NUMBERS
RELOCATABLE SUBROUTINES CALLED
OBJECT PROGRAM DATA TABLE
XXXXX STORAGE POSITIONS

The absence of one of the first three messages indicates that no undefined or unreferenced statement numbers have been found, or that no relocatable subroutines have been used by name in the source program. Undefined statement numbers are those referenced by a transfer, DO or I/O statement, but not defined.

After the final summary has been typed, the following message will be typed on the console typewriter and the program will halt.

PROCESSING COMPLETE

Pressing the start key causes the program to clear the symbol table compiled during the procesing of the previous source program and to prepare to process a new program. When this has been done, the program will type the message which calls for the entry of the source program.

General Make-up of Program Deck

The 1620 FORTRAN Pre-Compiler program deck is made up of four sections, identifiable by sequence number as follows:

00000 - 00001	Loading routine
00002 - 00348	Pre-Compiler program
00359 - 00365	Arithmetic tables
02001 - 02007	Relocatable subroutine data

The relocatable subroutine data consists of a card containing the number of relocatable subroutines included in the program deck, and the following cards contain the names of these subroutines. These cards must be in the proper sequence. If they are out of sequence, an error message will be typed and the 1620 will halt. Restore the sequence of these cards (including the two cards from the reader stacker), press the reader start and start keys to continue the operation.

The relocatable subroutine cards are punched with the function name starting in column 1 and the sequence number in columns 76 through 80. Subroutine names added to the system must be punched in the same manner.

General Make-up of Program Tape

The 1620 FORTRAN Pre-Compiler tape consists of a loading routine which loads the multiply and add tables, and the program which follows. The last seven records are the relocatable subroutine data containing the number and names of the relocatable subroutines included in the program. These records **must** be exact duplicates of the corresponding records which are included in the 1620 FORTRAN processor tape. Additions to the list of relocatable subroutines in the system must be made to the Pre-Compiler tape in exactly the same form as prescribed for the 1620 FORTRAN processor tape.

Tape Data

For the purpose of tape identification, a title and data message have been incorporated in the 1620 FORTRAN Pre-Compiler tape. The first two records of the tape contain the title and data information. After these records have been read into the 1620, the following message will be typed:

1620 FORTRAN PRE-COMPILER 11/15/61

Normal processing continues after the message has been typed.

Duplicating the Pre-Compiler Tape

The FORTRAN Pre-Compiler tape may be duplicated and/or modified by the use of the program for duplicating the FORTRAN processor and subroutine tapes, in the manner described for duplicating the processor tape.

Appendix A — Summary of 1620 FORTRAN Statements

ACCEPT

Format: "ACCEPT *n, List*"
where *n* is the statement number of a FORMAT statement and *List*, is a list of the quantities to be typed.

Purpose: This statement causes the program to read information from the console typewriter in accordance with FORMAT statement *n* and to transmit this information into core storage as the values of the variables in the list.

Example: ACCEPT 30, A, B, C, D(3)

ACCEPT TAPE

Format: "ACCEPT TAPE *n, List*"
where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be entered.

Purpose: This statement causes the program to read data from the tape reader and transmit this data into core storage as the values of the variables in the list.

Example: ACCEPT TAPE 48, K, A(J)

CONTINUE

Format: "CONTINUE"

Purpose: This statement is used as the last statement in the range of a DO when the DO would otherwise end with an IF or GO TO statement.

Example: CONTINUE

DIMENSION

Format: "DIMENSION *v(d), v(d, d), v(d)*"... for one- and two-dimensional arrays.

where each *v* is the name of a variable followed by parentheses enclosing one or two constants, *d*, represents the number of elements in each array.

Purpose: The DIMENSION statement provides information for the processor necessary for the allocation of storage in the object program for the elements of arrays of quantities.

Example: DIMENSION A(10), B(10,5)

DO

Format: "DO *n* $i = m_1, m_2, m_3$"
where *n* is a statement number, *i* a fixed point variable, and m_1, m_2, and m_3 can be either a fixed point constant or a fixed point variable.

Subscripts and sign indication are not permitted in a DO statement. If m_3 is not stated, it is taken to be 1.

The commas are required punctuation.

Purpose: The DO statement simplifies the programming of loops and provides greater flexibility in looping.

Example: DO 20 JBNO = 1, 10

END

Format: "END"

Purpose: The END statement is a signal to the compiler that the end of the source program has been reached.

Example: END

FORMAT

Format: "FORMAT $(s_1, s_2, s_3, \ldots, s_n)$"

where s_1, s_2, s_3, and s_n are specifications.

Purpose: This statement describes the type of conversion and format of data to be used in the transmission of input/output lists.

Example: 2 FORMAT (I2/F10.4,E12.4)

GO TO

Format: "GO TO n"

where n is a statement number.

Purpose: This statement interrupts the sequential execution of statements; it specifies the number of the next statement to be performed.

Example: GO TO 30

Computed GO TO

Format: "GO TO $(n_1, n_2, \ldots, n_m), i$"
where $n_1, n_2, \ldots, n_m$ are statement numbers and i is a fixed point variable. The variable cannot be subscripted.

Purpose: The computed GO TO statement transfers the program to the 1st, 2nd, etc., statement number in the list depending upon whether the value of i is 1, 2, . . . , etc.

Example: GO TO (3, 4, 6), L

IF

Format: "IF (a) n_1, n_2, n_3"
where a is an expression and n_1, n_2, and n_3 are statement numbers.

Purpose: The IF statement transfers the program to a particular statement depending upon the value of an expression.

Example: IF (A-B) 10, 5, 7

IF (SENSE SWITCH)

Format: "IF (SENSE SWITCH i) n_1, n_2"
where i is the number of one of the console program switches, and n_1 and n_2 are statement numbers.

Purpose: This statement transfers the program to a particular statement depending upon the setting of any one of the four console program switches.

Example: IF (SENSE SWITCH 3) 14, 50

PAUSE

Format: "PAUSE"

Purpose: The PAUSE statement is used as a convenient means of causing the **object** program to halt **temporarily.** Pressing the start switch causes the program to resume with the statement following the PAUSE statement.

Example: PAUSE

PRINT **(See TYPE)**

PUNCH

Format: "PUNCH *n, List*"
where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be punched.

Purpose: This statement causes the items in the list to be punched in cards in the format specified by the statement *n*.

Example: PUNCH 1, A, D, C

PUNCH TAPE

Format: "PUNCH TAPE *n, List*"
where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be punched.

Purpose: This statement causes the items in the list to be punched into paper tape in the format specified by statement number *n*.

Example: PUNCH TAPE 4, A, B, C

READ

Format: "READ *n, List*"
where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be transmitted.

Purpose: This statement causes data to be read from a card in the 1622 Card Read Punch.

Example: READ 4, A, B, C

TYPE

Format: "TYPE *n, List*"
"PRINT *n, List*"
where *n* is the statement number of a FORMAT statement and *List* is a list of the quantities to be typed.

Purpose: This statement causes the quantities in the list to be typed on the typewriter in accordance with FORMAT statement *n*.

Example: TYPE 4, A, B, C

STOP

Format: "STOP"

Purpose: This statement causes the computer to halt during the processing of the object program, return the typewriter carriage and type the word "STOP."

Example: STOP

Appendix B — Summary of 1620 Operating Principles

Typewriter Input

The typewriter is part of the 1620 console and is used for both input and output.

Input

The typewriter is used to enter both data and instructions directly into core storage. Pressing the console insert key unlocks the keyboard and permits data to be entered into core storage starting at location 00000. Each depression of a typewriter key enters the character into core storage one location higher than the previous character. As many as 100 characters can be entered from the typewriter. After the 100th character is entered, an automatic release is initiated and the machine returns to manual mode.

When less than 100 characters are entered, entry of the last desired character should be followed by pressing the console release and start keys, or by pressing the R-S key on the typewriter keyboard. The R-S key combines the release and start functions of the console keys. The R-S symbol is typed as a permanent record that the R-S key has been used.

Programmed selection of the typewriter unlocks the keyboard and leaves the computer in automatic mode for manual entry of data on the typewriter. Data entry starts at the addressed location (P address) of the instruction and enters core storage at successively higher-order positions until the release key is depressed.

If a record mark is required in core storage following the last character entered, the record mark key on the typewriter must be pressed before pressing the release key on the console.

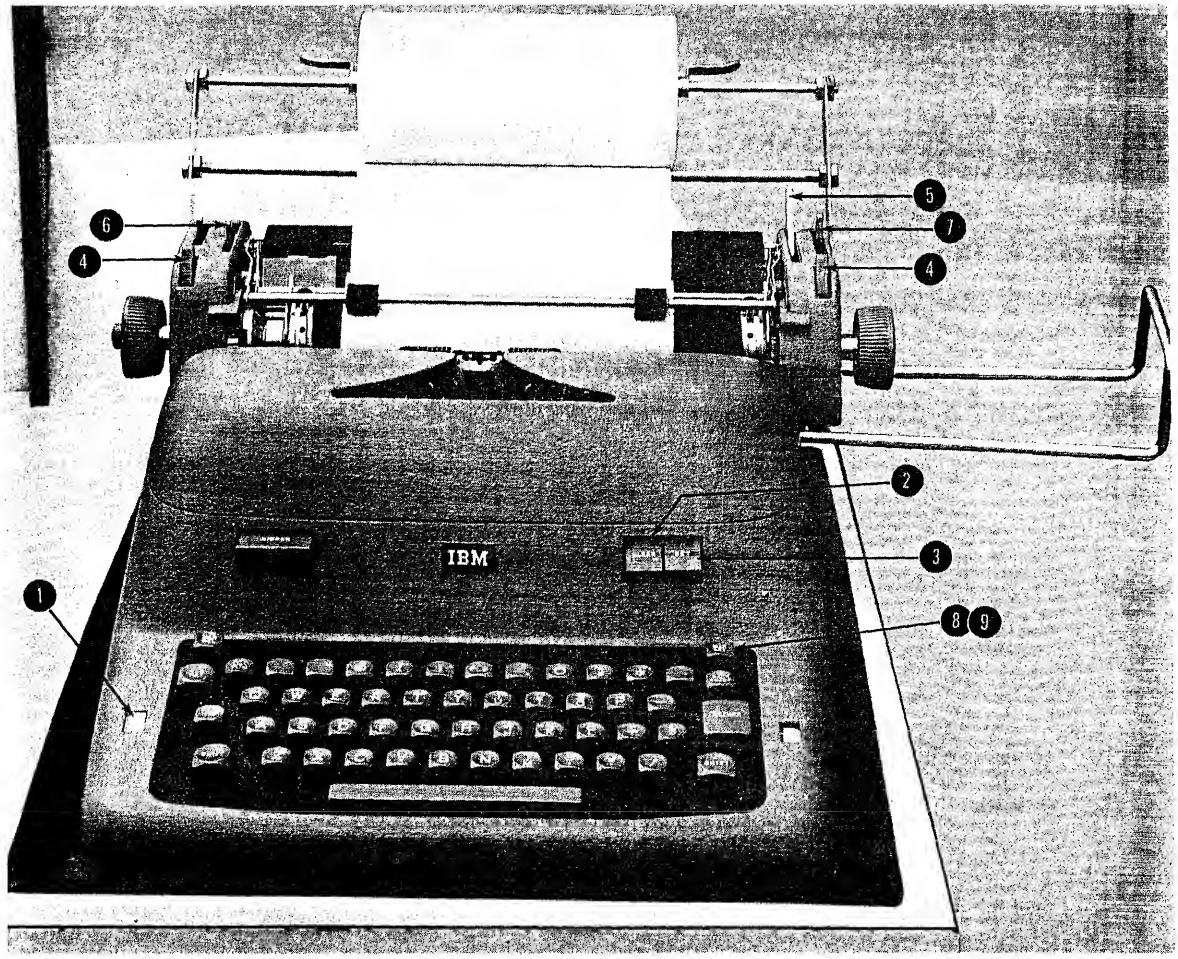


Figure 1. IBM 1620 I/O Typewriter

Pressing the console release key relocks the keyboard and gives the computer an end-of-i/o indication. No record mark is entered into core storage by pressing the release key.

Output

The typewriter prints data from core storage when programmed to do so. When the right-hand margin is reached, the carriage returns automatically and typing continues until a record mark is sensed or until the release key is pressed.

Manual Adjustments to Typewriter

(1) *Impression Indicator.* To determine the force with which the type bars strike the paper, position the lever under this window for settings from 0 to 10. The higher the indicator setting, the harder the type bars strike.

(2) *Tab Clear Lever.* To clear tab stops, tabulate to the point to be cleared and press the clear lever. To clear all stops at once, position the carriage at the right margin, hold down the clear lever, and return the carriage to the left margin stop.

(3) *Tab Set Lever.* To set tabular stops, move the carriage to the desired position and press the set lever. Set tab stops only when the indicator pointer is in line with a white marking on the front paper scale below it.

(4) *Carriage Release Lever.* Press the lever on either side to free the carriage then manually move the carriage to the right or left.

(5) *Paper Release Lever.* To free the paper for positioning or quick removal, move this lever forward.

(6) *Line Space Lever.* Moved to position 1, 2, or 3, the line space lever provides for single, double, or triple line spacing, respectively.

(7) *Multiple Copy Control.* This lever moves the platen backward to compensate for the greater thickness of additional copies. As a general rule the lever should be set at "A" for one to three copies and moved back one position for each additional three to five copies.

(8) *Left-Hand Margin Set.* The left margin stop is set as follows:

1. Return the carriage to the present left margin stop.
2. Press the margin set key.
3. Manually move the carriage as near as possible to the position desired. The backspace key and space bar are convenient to use to obtain the exact position desired, with the margin set key depressed.
4. Release the margin set key.

(9) *Right-Hand Margin Set.* The right margin stop is set as follows:

1. Move the carriage to the left until stopped by the right margin stop.
2. Press the margin set key.
3. Move the carriage right or left to the desired position.
4. Release the margin set key.

Paper Tape Input

Data is punched and read as holes in a 1-inch-wide chad paper tape (in chad paper tape the holes are completely punched out) at a density of ten characters to the inch. Eight-track paper tape code is used. Seven positions, or tracks, across the width of the tape, are used for the coding of numerical, alphabetic, and special characters. One track is used for EL (end-of-line) characters. Figure 2 represents a section of paper tape, which illustrates the eight tracks and all coded characters.

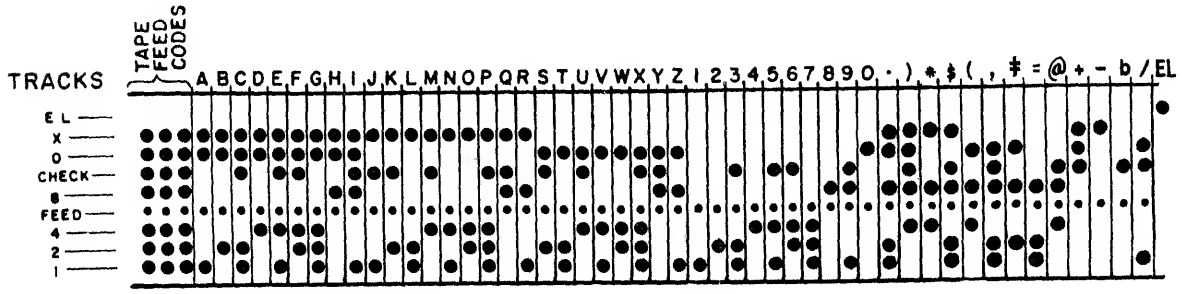


Figure 2. Paper Tape Tracks and Codes

1624 Tape Punch

The tape punch (Figure 3), housed below the tape reader in the IBM 1621, punches data from core storage into paper tape at the rate of 15 characters per second. The characters are sent serially from core storage starting with the location addressed by an output instruction. Each character is translated to 8-track code before being punched.

If a character with incorrect parity is transmitted from core storage and punched, or if a valid character is incorrectly punched, the tape feed does not advance. The computer stops in both the automatic and manual mode; the automatic and manual lights and the punch no feed and write check lights on the 1620 console are turned on. Functions of these lights are described under CONSOLE. Program processing can be resumed with the following procedure:

1. Position the 1624 tape feed switch ON.
   a. The feed code (all punches) is punched over the incorrect character.
   b. The punch no feed and write check lights are turned off.
   c. The machine is returned to manual mode only.
2. Press the start key on the 1620 console.
   a. The original character from storage is again punched. If an incorrect character still persists, the record may be corrected, if desired, before processing continues.
   b. The computer continues processing.

If the 1624 runs out of paper tape, the machine stops in automatic mode and the punch no feed light turns on. The "character correction procedure" outlined is used to resume operation.

Figure 3. IBM 1624 Tape Punch

Loading the Tape Punch

Place the roll of unpunched tape on the turntable and thread as shown in Figure 3. The tape retainer (F) must be rotated to the left by pushing back on its extended left edge. This moves the tape lever (D) forward to facilitate threading. An unwound section of tape is then threaded as follows:

1. Through tape guide (A).
2. Inside tape guide (B).
3. In front of tape tension guide (C).
4. In back of tape lever (D).
5. Between the punching mechanism and the punch guide block (E), which can be seen in front of the tape.
6. Between the guides on the tape retainer (F). With the end of the tape held to the left, the tape retainer (F) is returned to normal position, which causes the pins on the feed roll to pierce through the blank tape. The tape lever simultaneously returns to normal position with the top guide above the tape.

The tape feed key is used to repetitively punch automatic feed punches and to provide a leader section of paper tape. Approximately 60″ of leader is needed for threading paper tape on the 1621 and can be obtained from the 1624 in 40 seconds. The leader is threaded into the 1624 take-up reel so that the top edge of the tape is at the outside of the reel.

1621 Paper Tape Reader

The paper tape reader reads coded alphameric characters from 8-track paper tape at the rate of 150 characters per second. The characters are photoelectronically sensed and placed in core storage. If a parity error is sensed, the read check indicator (console panel) is turned on. The computer remains in automatic mode and continues to read until the end-of-record indication (a hole in the EL channel) is reached. Whether the computer stops, depends upon the setting of the I/O check switch. The end-of-record signal causes a record mark to be placed in core storage as the rightmost digit of the input record.

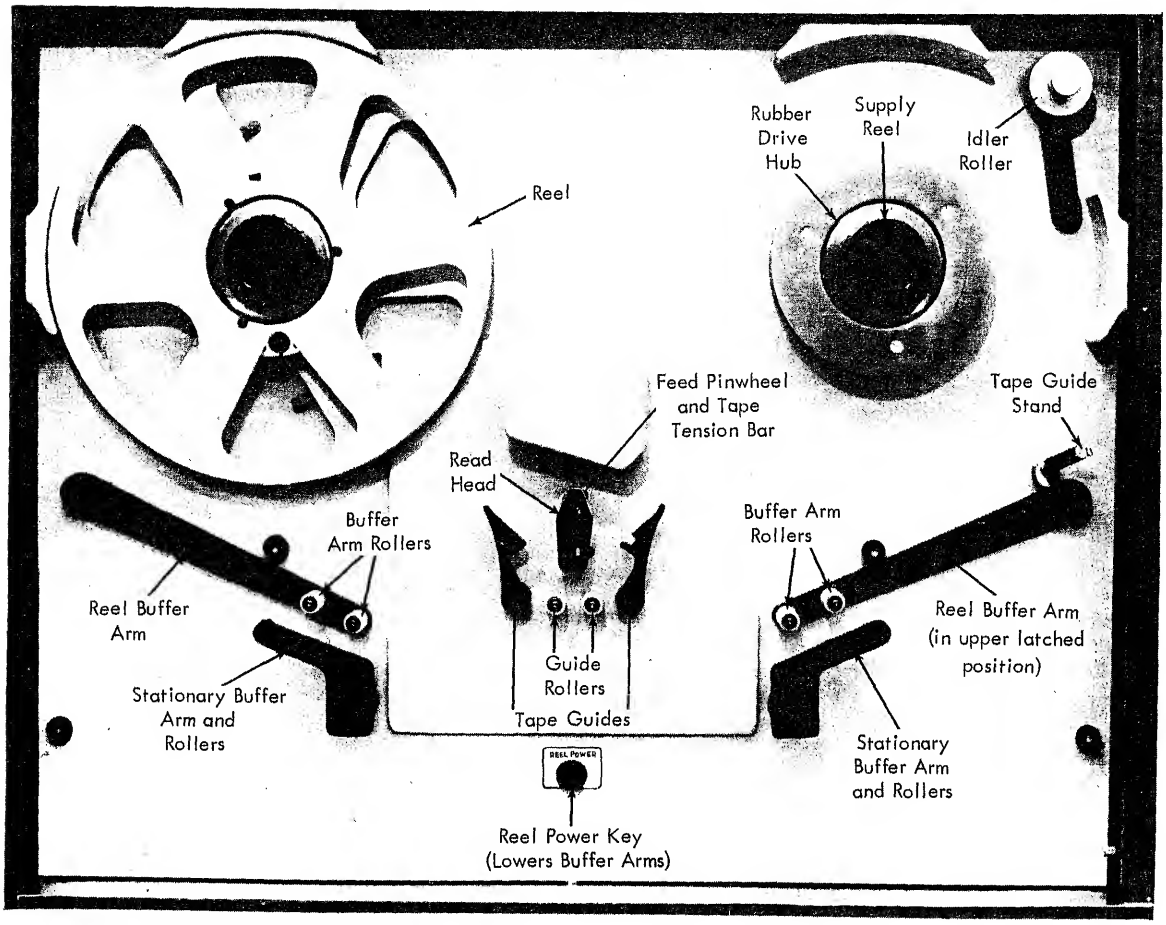


Figure 4. IBM 1621 Tape Loading Area

Loading the Paper Tape Reader

Paper tape can be handled in 2 forms. The procedure for loading each one varies slightly. The names of machine components used in the following descriptions of loading procedures are given in Figure 4.

CENTER ROLL FEED

The center roll feed eliminates the necessity for rewinding paper tape rolls which would expose the starting end of the tape on the outside of the tape roll. Figure 5 shows that tape is supplied from the inside of the center roll feed, to the supply reel, around the read head, and onto the take-up reel.

The procedure for loading paper tape from the center roll feed is as follows:

1. Position the reel strip switch to REEL.
2. Place the reel buffer arms in the upper latched positions.
3. Open the tape guides and form an inverted U (∩) with the center section of the first eight feet of paper tape. Wrap the paper tape around the read head with sufficient tension to keep the runout and tape tension contacts closed. Start on the take-up reel side of the read head. Run a finger up over the tape on top of the read head, smoothing the tape down with a firm, moderate pressure so that the tape tension bar is slightly depressed and the right side of the feed pinwheel engages the tape feed holes. Be careful not to tear the feed holes. The tape feed holes must mesh with both sides of the pinwheel.
4. Close the tape guides.
5. Thread the leading section of paper tape under the guide roller, between the stationary buffer rollers and buffer arm rollers, and onto the take-up reel, as shown in Figure 5.
6. Thread the paper tape from the right side of the read head under the guide roller, between the stationary buffer rollers and buffer arm rollers, over the supply reel (the rubber drive hub must

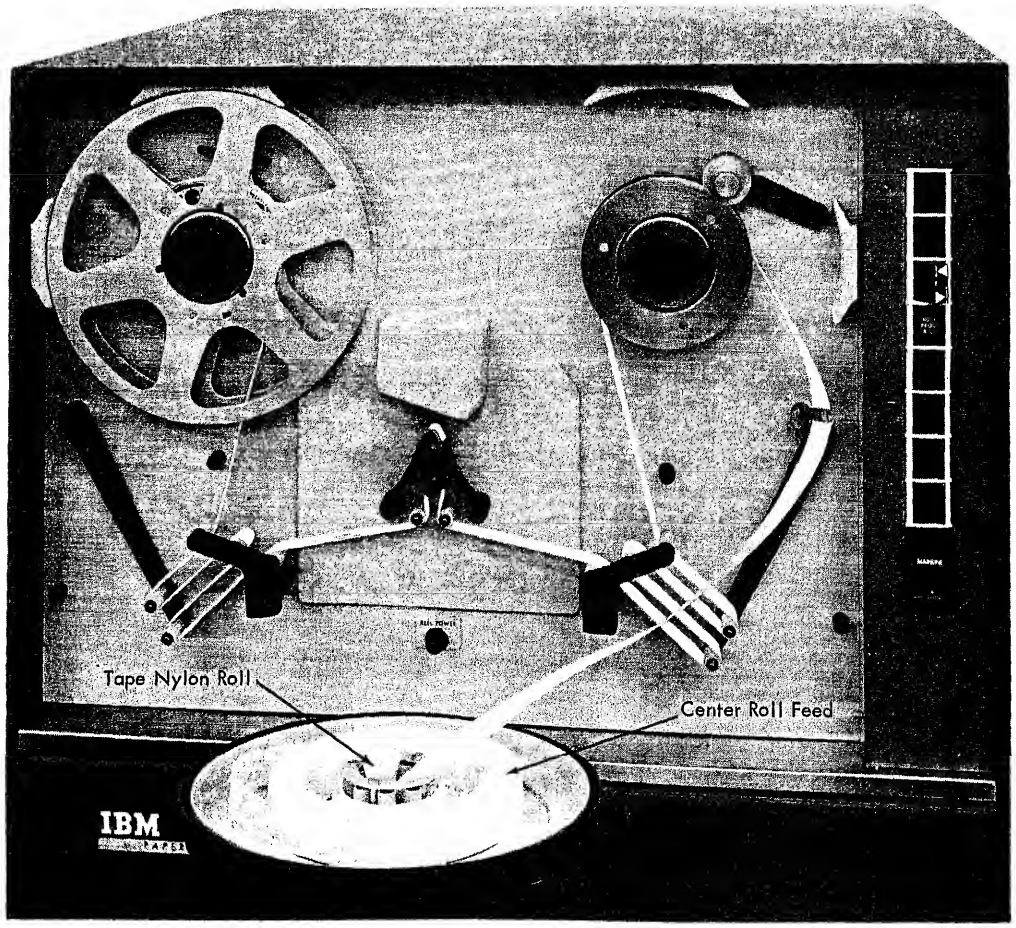


Figure 5. Center Roll Feed Loaded

be installed), around the tape guide stand, and around the tape reel nylon roll.

7. Lower the idler roller onto the supply reel.
8. Lower the buffer arms gently.
9. Press the reel power key. The buffer arms should swing down to a neutral position, applying tension to the paper tape.

NOTE: The roll of paper tape must be positioned centrally, or evenly, around the center rollers to prevent excessive vibration during reading.

REEL

A reel of paper tape may be read on the 1621 by removing the rubber drive hub from the supply reel and by mounting the reel of tape in its place. The tape is threaded from the right-hand side of the reel directly to the stationary buffer rollers, and then to the take-up reel as described under CENTER ROLL FEED. Figure 6 shows a reel of tape threaded on the 1621.

Operating Switches and Lights

The following switches and lights are used in the operation of the 1621.

Power Switch. With this switch on, all necessary power for operation of the 1621 is supplied by the 1620.

Reel Strip Switch. In reel mode, tape is fed from the supply reel then to the left onto the take-up reel.

Reel Power Key. Pressing this key operates the supply and take-up reels to position the paper tape for reading and to place the machine in ready status.

Nonprocess Runout Key. Pressing this key causes paper tape to feed. Ready status is terminated and all data transfer is blocked until all paper tape has passed. Paper tape must be reloaded and the reel power key pressed before the machine can be returned to ready status.

Power On Light. This light ON indicates that power is supplied from the 1620.

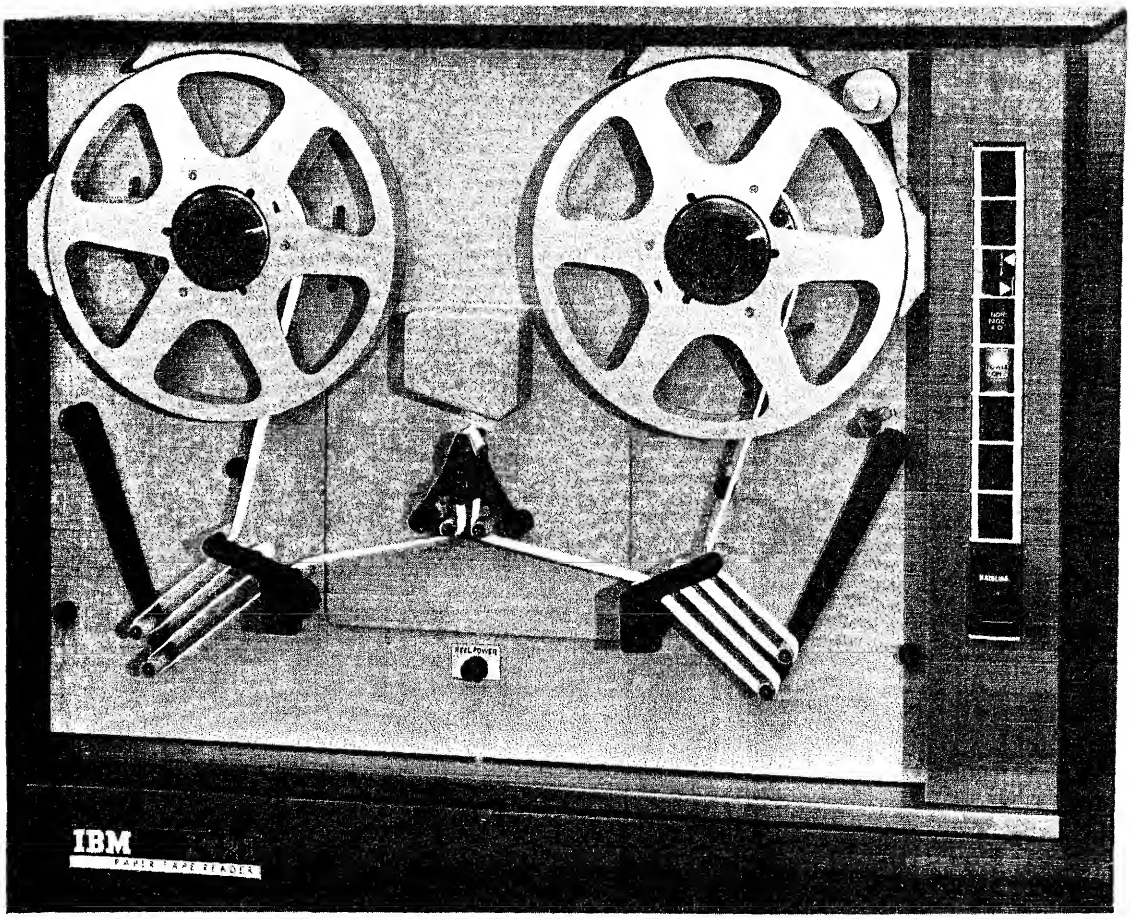


Figure 6. Paper Tape Reel Loaded

Card Input

1622 Card Read Punch

The IBM 1622 Card Read Punch (Figure 7) provides punched card input and output for the IBM 1620 Data Processing System. The reader and punch feeds are separate and functionally independent, with individual switches, lights, checking circuits, buffer storage, and instruction codes. Under program control, up to 250 cards per minute can be read and 125 punched. Reading, punching, and processing can occur simultaneously because of individual buffer storage. Buffer storage data is transferred in 3.4 milliseconds; the remainder of the reader and punch feed cycle time is available for processing.

As shown in Figure 8, cards are fed from the read hopper on the right and the punch hopper on the left. Each hopper has a capacity of 1,200 cards. Both feeds have misfeeding and jam detection, and a select and nonselect stacker. The 1,000-card-capacity stackers are of the radial type: the cards are stacked on end to permit their removal while the 1622 is running.

CARD READER AND PUNCH DRIVE MOTORS

If either the read or punch feed is not used for approximately one minute, the drive motor for that feed is turned off to reduce noise and wear. However, the 1622 is still in ready status and will respond to a read or write command.

Card Read

Cards are fed 9-edge first, face down, past two reading stations, check and read. Input buffer storage is initially loaded with 80 columns of card data during the start key or load key run-in operation. Thereafter, each card feed cycle is under program control.

Card Punch

Cards are fed 12-edge first, face down, past the punch and check stations.

Operator Keys and Lights

The card reader and card punch have separate keys and lights (see Figures 7 and 8).

CARD READER

Reader On/Off Switch. The reader on/off switch is used to supply power to the reader and to turn on the power ready light. The 1620 power on/off switch must be on to make the 1622 reader on/off switch active.

Load Key. The load key causes data from the first card to be checked, read into buffer storage, and automatically transferred in numerical mode to core storage positions 00000 through 00079. Upon completion of this data transfer, another card feed cycle occurs which loads buffer storage with data from the second card. The 1620 then simulates release and program start at 00000. The instructions from the first card, now in 00000 through 00079, can be used to continue loading the program or to begin processing. The 1620 must be reset and in manual mode to make the load key operate correctly.

Start Key. The start key is used (1) to run in cards, which are then placed under program control (data from the first card is checked and loaded in input buffer storage); (2) to set up a runout condition, which permits programmed reading of the cards remaining in the

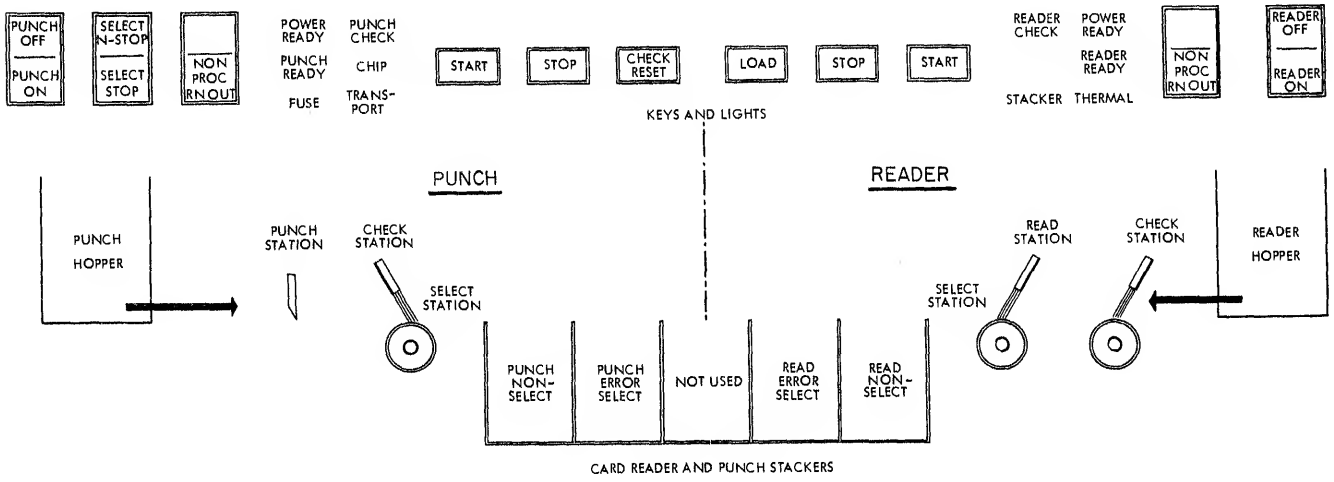


Figure 7. Schematic Diagram of 1622 Keys, Lights, and Card Feeds

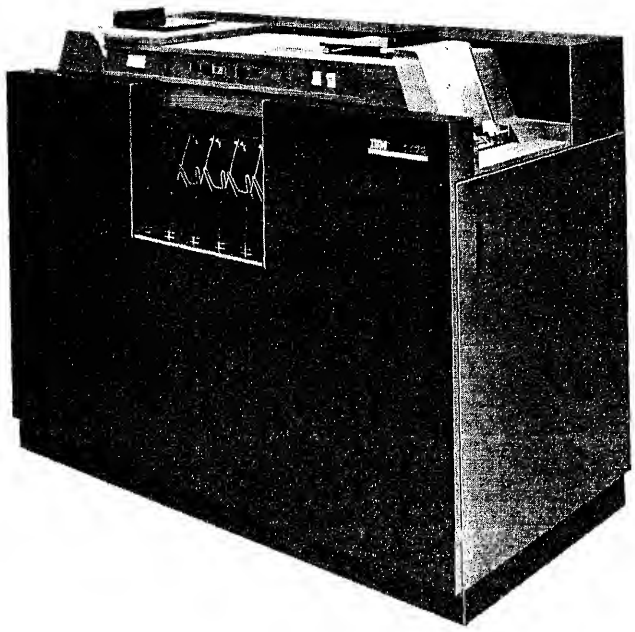


Figure 8. IBM 1622 Card Read Punch

feed when the hopper has become empty; and (3) to restore ready status after the reader has been stopped by either the stop key, an empty hopper, an error, a misfeed, or a transport jam.

Stop Key. The stop key is used to stop the read feed at the end of the card cycle in progress and/or to remove the reader from ready status. Data that is entered into buffer storage during the read cycle in progress is transferred to core storage. The computer continues processing until the next read card command causes a reader no feed stop.

Nonprocess Runout Key. The nonprocess runout key is used to run cards out of the read feed after a reader check error, or after the stop key has been used to stop the reader. The cards are run out into the read select stacker without a buffer storage to core storage transfer. The reader check light and check circuits are turned off. Cards must be removed from the hopper to make the nonprocess runout key active.

Reader Ready Light. The reader ready light is turned on to indicate that the first card has been loaded into buffer storage with the start key, without a reader check error. It remains on until the following occurs: a depression of the stop key, a reader check error, a transport jam, a misfeed, or an empty hopper.

Reader Check Light. The reader check light is turned on by an unequal comparison between the read and check stations and by incorrect parity detected in buffer storage during card read. When there is an unequal comparison, the reader is stopped, ready status is terminated, and the buffer storage data just read cannot be transferred to core storage on the next read command.

1620 Console Read Check Light. The 1620 read check (06) indicator and console read check light are turned on by a 1620 parity error during a buffer storage to core storage transfer.

1620 Console Reader No Feed Light. The console reader no feed light is turned on each time the reader is selected by a read command. The light remains on, if for any reason the reader is not in ready status and the read command therefore cannot be executed. It appears to be on almost continuously when the time between read calls is less than 240 ms, indicating that processing time is available.

CARD PUNCH

Punch On/Off Switch. The punch on/off switch is used to supply power to the punch and to turn on the power ready light. The 1620 power on/off switch must be on to make the 1622 punch on/off switch active.

Start Key. The start key is used to feed cards to the punch station initially or after an error and nonprocess runout, and to re-establish ready status after an empty hopper, a misfeed, a transport jam, or a stop key depression.

Stop Key. The stop key is used to stop the punch feed at the end of the card cycle in progress and/or to remove the punch from ready status.

Check Reset. The check reset key is used to reset error circuits and turn off the punch check light. A start key or nonprocess runout key depression follows.

Select N Stop — Select Stop Switch. This switch is used to control the stopping of the punch when error cards are selected into the punch error select stacker. With the switch set to STOP, the punch feed stops with the error card in the select stacker.

Nonprocess Runout Key. Following a punch check error, pressing of the nonprocess runout key resets the error circuits and causes the punched card that is between the punch station and the punch check station, if it is in error, to follow the error card into the select stacker. If this card is in error, the punch check light is turned on again. The next two (blank) cards go into the nonselect pocket. These cards should be removed before further processing.

This key is also used to run out and check the last punched card of a job. Cards must be removed from the hopper to make the nonprocess runout key operative.

Punch Ready Light. The punch ready light is used to indicate that the 1622 has a card in punch position and will respond to a write command from the 1620. The ready light is turned off by a punch check error, an empty hopper, a full chip box, a stop key depression, a transport jam, or a misfeed.

Punch Check Light. The punch check light is turned on when there is an unequal comparison between the data punched and the data read (one card feed cycle later, at the check station), or when a 1622 parity error occurs during punching (select stop switch set to STOP). The machine stops, and ready status is terminated.

Chip Light. The chip light is turned on to indicate that the chip box should be emptied.

1620 Console Write Check Light. The 1620 write check (07) indicator and console light are turned on by a parity error during a core storage to buffer storage transfer. The 07 indicator may be programmed to transfer data several times and to halt if a correct transfer cannot be obtained.

1620 Console Punch No Feed Light. The console punch no feed light is turned on each time the punch is selected by a write command. The light remains on until the punch unit is ready and executes the command. Normally, no light is seen if commands are further apart than 480 milliseconds. The write command cannot be executed until the punch is in ready status.

CARD READER/PUNCH LIGHTS

The stacker, transport, fuse, and thermal lights are used commonly by both the read and punch feeds as follows:

Stacker Light. The stacker light is turned on when a stacker is full. Both feeds are stopped temporarily and removed from ready status. The ready light remains on. Operation resumes automatically after the stacker is emptied.

Transport Light. The transport light is turned on when a card jam has occurred in either the read or punch feed or above any stacker. When this occurs, both feeds are stopped and removed from ready status. Both start keys must be pressed to resume operation after the condition is corrected.

Fuse Light. The fuse light turns on to indicate a blown fuse.

Thermal Light. The thermal light is turned on if the internal temperature of the 1622 becomes excessive. After several minutes delay, the 1620 console reset key may be pressed to turn off the thermal light. If pressing the reset key turns off the thermal light, the 1620 power switch must be turned off and then on again. Operation may be resumed after the power ready light is turned on.

Console

The console (Figure 9) is an integral part of the central processing unit and provides for manual or automatic control of the system. The console lights, keys, switches, and typewriter are used to:

Instruct the machine manually.
Display machine and program status indicators.
Display the contents of core storage and registers.
Place data and instructions in core storage.
Alter the contents of core storage.
Alter machine functions.

Keys, Indicator Displays, and Switches

Small incandescent lights are used to represent the on and off conditions of internal check indicators.

Seven console switches (four program and three machine check switches) are provided to externally control the execution of machine functions for which two

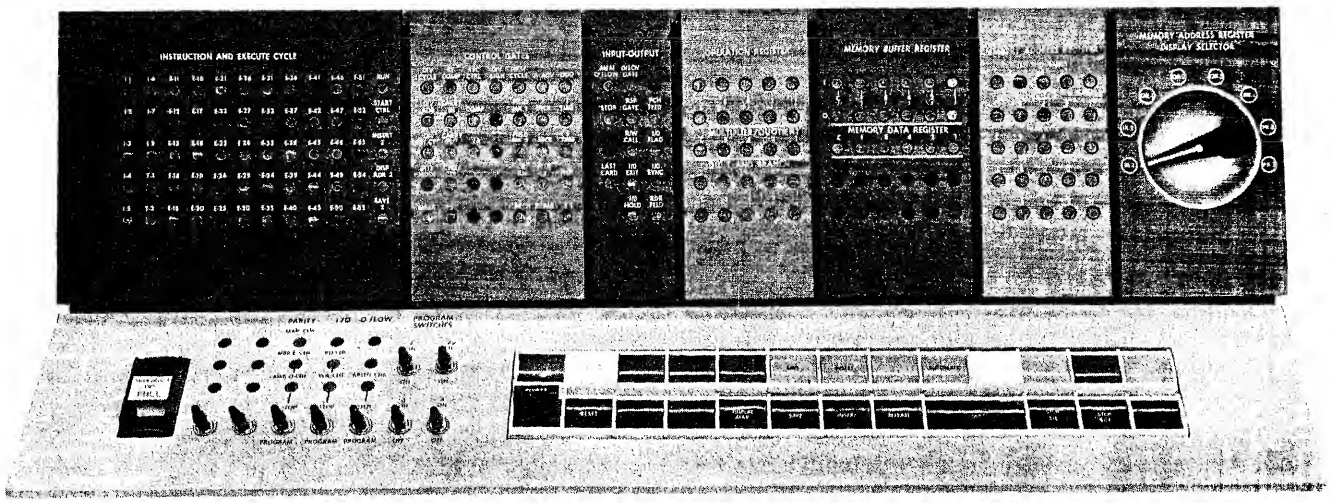


Figure 9. IBM 1620 Console

alternative logic paths are provided. One or the other of the paths is selected, depending upon the setting of the appropriate switch.

Machine Check Indicators and Switches

Machine operation may be altered by the condition of a machine check indicator and an associated check switch (Figure 10). An indicator that is turned on causes the computer to halt if the associated check switch is set to STOP, or to continue in automatic mode if the associated check switch is set to PROGRAM. Regardless of the check switch setting, the associated check light provides a visual sign of the indicator status.

Pressing the reset key turns all check indicators and lights off. Parity, I/O, and overflow check indicators are provided.

PARITY CHECK INDICATORS

Internal data flow errors are recorded by the parity check indicators: MBR-E and MBR-O. Normally, the parity check switch is set to STOP.

MBR-E (Memory Buffer Register-Even) Check Light. This light and indicator are turned on when the digit in the even address portion of the MBR has a parity error.

MBR-O (Memory Buffer Register-Odd) Check Light. This light and indicator are turned on when the digit in the odd address portion of the MBR has a parity error.

MARS (Memory Address Register Storage) Check Light. This light turns on when a digit in MARS has a parity error. This is an unconditional machine stop and is not affected by the position of the parity check switch.

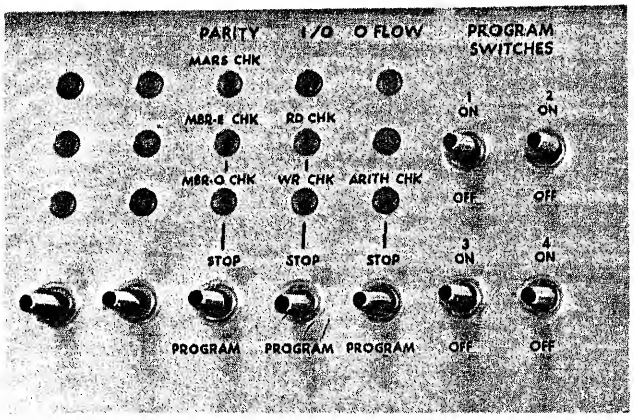


Figure 10. Indicator Displays and Switches

INPUT/OUTPUT (I/O) CHECK INDICATORS

RD CHK (Read Check) Light. This light and indicator are turned on when an input character with a parity error is detected prior to conversion of input data to BCD code.

WR CHK (Write Check) Light. This light and indicator are turned on when an output character with an even number of bits is detected during conversion of output data from BCD to output code.

OVERFLOW ARITH CHK (ARITHMETIC CHECK) INDICATOR

An overflow that occurs as a result of an add, subtract, divide, or compare operation turns on the overflow check indicator and light.

CONSOLE PROGRAM SWITCHES

There are four modifier switches in this group. They are labeled PROGRAM SWITCHES on the console and are numbered 1 through 4.

REGISTER DISPLAY INDICATORS AND SWITCHES

The console panel displays the contents of registers by means of small incandescent lights, used to represent the bits present in each digit of a register (Figure 11). Each light, representing a particular bit position, is on only when its corresponding bit is present in the digit displayed.

Memory Buffer Register (MBR). The two stored digits affected by a core storage address (previously explained under TWO-CHARACTER TRANSFER) are displayed in the MBR. When the core storage location addressed for display is an even-numbered address, the digit at this location is placed in the MBR display in the E (even line); the O (odd) line contains the digit in the next higher-numbered location. If the core storage location addressed for display is an odd-numbered address, the digit at this location is placed in the MBR display on the O line; the E line contains the digit in the next lower-numbered location. When the machine is in alphabetic mode, the complete 2-digit representation of an alphameric character may be viewed at one time.

Memory Data Register (MDR). One line of six indicator lights displays the bit configuration of each digit in core storage as it is read out. These digits can be seen on single cycle operation by using the SCE key (described under CONTROL SWITCHES, KEYS, AND SIGNAL LIGHTS). The digit displayed in the MDR display is duplicated in the MBR-even or MBR-odd display, depending on whether the digit read out is located at an even or an odd numbered core storage position.

Operation (OP) Register. Two lines of five lights each display the bit configuration of the two digits representing the operation code of the instruction last executed. Flag bits of these two digits are not displayed.

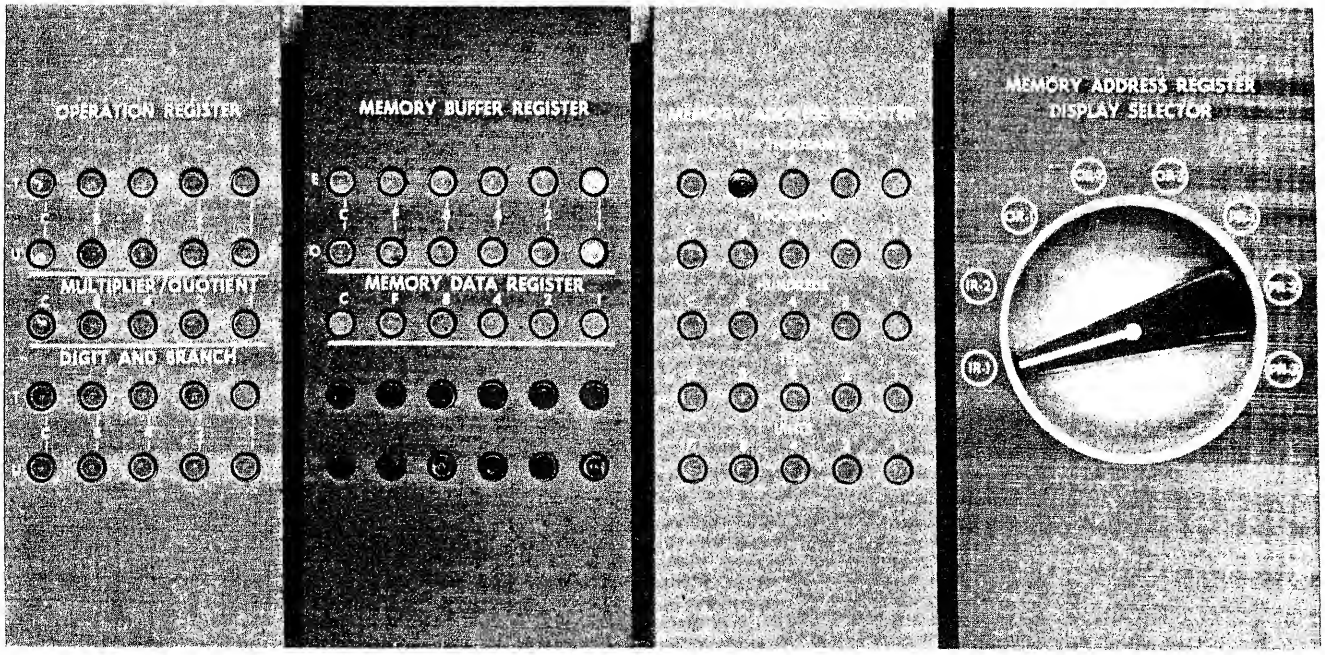


Figure 11. Register Display Indicators

Sense and Branch (S-B). Two lines of five lights each display Q_8 and Q_9 of the Branch Indicator, Branch No Indicator, and Input/Output instructions, from the Sense and Branch register. Input/output device codes (digits 01-05) are displayed for input/output and control instructions.

Digit Register. These two lines of six lights each are used primarily for diagnostic testing by customer engineers.

Multiplier. This 5-light multiplier register display shows each multiplier digit as it is used during a multiply operation.

Memory Address Register (MAR). Five lines of five indicator lights each display the bit configuration of the five-digit address in any one of the eight MARS entry. Signal lights associated with the control keys provide a visual indication of a specific operating condition of the computer and indicate which step of the keying procedure was last completed.

POWER ON/OFF SWITCH — POWER ON LIGHT

The power on/off switch has an ON and OFF position. Set to the ON position, it applies electrical power to the computer and turns on the power on light.

POWER READY LIGHT

The power ready light comes on when internal machine temperature and voltages reach proper operating values. There is a delay from the time the power on/off switch is positioned ON until operating temperature and voltages are obtained. This delay varies with room tem-

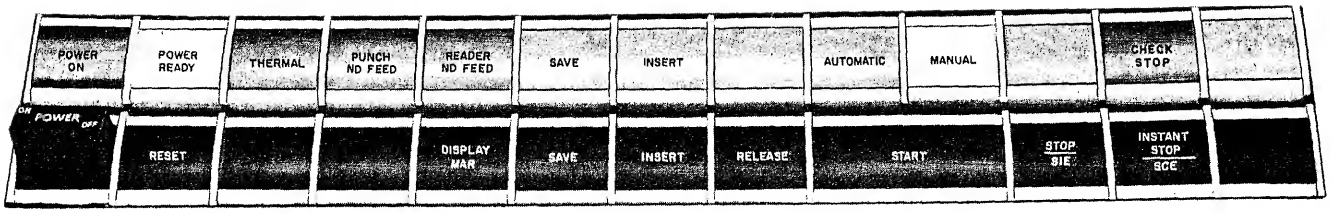


Figure 12. Control Keys and Signal Lights

The automatic light, when on, indicates that the computer is in automatic mode (e.g., while executing a stored program or while entering data into core storage from the typewriter keyboard).

Manual mode is initiated and the manual light is turned on by the execution of a Halt instruction or by pressing the release key (on an I/O operation only), instant stop key, or stop key. Pressing the start key, insert key, or display MAR key initiates automatic mode and turns the manual light off. The save light and/or the no feed light can be on when the manual light is on.

Both the manual and automatic lights are on when an instruction is single-cycled with the SCE key.

RESET KEY

The reset key is used to restore all machine status indicators, machine check indicators, and signal lights to their initial or reset condition. The reset key functions only when the computer is in the manual mode (manual light on). Parity errors can occur if the reset key is used while the computer is in the automatic mode. When the computer is in the automatic mode, the instant stop key should be pressed to put the computer in the manual mode and permit use of the reset key.

INSERT KEY AND INSERT LIGHT

Pressing the insert key places the 1620 in automatic mode. Pressing the insert key also turns on the insert light and activates the typewriter keyboard so that direct entry of instructions may be made in numerical mode, starting at 00000 and continuing into higher-numbered storage positions. As many as 100 digits may be keyed in. After the 100th digit is entered, an automatic release is initiated and the 1620 returns to manual mode. Pressing the start key initiates computer operation beginning at 00000.

SAVE KEY AND SAVE LIGHT

Pressing the save key turns on the save light and saves the address of the next sequential instruction to be executed. This address is saved in Product Address Register 1 (PR-1).

RELEASE KEY

The release key is used to terminate any input/output operation, including console keyboard entry of data into core storage. When this key is pressed, manual mode is initiated, the manual light is turned on, and the insert light is turned off.

The release key is operative only when the computer is in automatic mode and performing an I/O operation.

STOP/SIE (SINGLE INSTRUCTION EXECUTE) KEY

Pressing the stop/SIE key stops the computer in manual mode at the end of the instruction being executed.

The stop/SIE key also serves as a single instruction execute key. Successive depressions of the key cause one instruction to be executed for each depression. The manual light remains on.

INSTANT STOP/SCE (SINGLE CYCLE EXECUTE) KEY

Pressing the instant stop/SIE key causes the machine to stop at the end of the 20-microsecond machine cycle in progress. Successive depressions of the key cause single machine cycles. Both manual and automatic lights remain on.

CHECK STOP LIGHT

The check stop light is turned on when the machine stops because of a parity check. One or more of the parity or I/O check indicators that caused the stop is also on. The check stop light is turned off when the check indicators are reset or the parity or I/O switch is set to PROGRAM.

DISPLAY MAR KEY

The display MAR key is operative only when the manual light is on and the automatic light is off. Pressing the display MAR key causes display of the MARS register to which the MARS display selector switch is set.

The rotary switch should not be turned while the display MAR key is pressed.

READER NO FEED LIGHT

The reader no feed light is turned on when the computer attempts a paper tape read or card read operation and the reader is not in the ready status.

PUNCH NO FEED LIGHT

The punch no feed light is turned on if one of the following conditions exists:

1. The computer executes a write instruction using the tape punch and there is no paper tape on the feed reel.
2. A parity check occurs while punching paper tape.
3. The paper tape supply is exhausted.
4. The card punch is not ready. This not ready status is often temporary on a card punch operation because the buffer is interlocked while the punch cycle is in process.

Any of these conditions stops the computer in automatic mode with both the automatic and punch no feed lights turned on. When a parity error occurs, the I/O write check light is also turned on. Pressing the release key disconnects the punch and puts the computer in manual mode. Pressing the reset key, while in manual mode, turns off the punch no feed and I/O write check light. Manual correction and restart procedures can begin after pressing the release and reset keys.

THERMAL LIGHT

The thermal light is turned on if the internal temperatures of the 1620, 1622, or 1623 become too high. Power is turned off, and the power ready light goes off. The thermal light may be turned off by pressing the reset key after the internal machine temperatures return to normal. The power switch must be turned off and on again before power can be applied to the machine.

EMERGENCY OFF SWITCH

This switch is for emergency use only. If positioned OFF, all power is turned off in the machine and the blowers that cool the electronic circuits are stopped. Damage to the machine may therefore result.

Index

IBM 1620 Printed in U.S.A. C26-5619-1

IBM®
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York